OpenMP Technical Report 10: Version 5.2 Public Comment Draft

This Technical Report is the public comment draft for the OpenMP Application Programming Specification Version 5.2 that improves the OpenMP API Version 5.1 features for target directives, user-defined mappers, and memory allocators. This version also refines the syntax of OpenMP directives to be more concise and consistent. The minus reduction and several existing instances of syntax that is inconsistent with the general OpenMP syntax format have been deprecated. See Appendix B.1 for the list of deprecated features and Appendix B.2 for the list of added features.

EDITORS

Bronis R. de Supinski

Michael Klemm

July 15, 2021

Expires November 10, 2021

We actively solicit comments. Please provide feedback on this document either to the Editors directly or by emailing to info@openmp.org

OpenMP Architecture Review Board – www.openmp.org – info@openmp.org
OpenMP ARB, 9450 SW Gemini Dr., PMB 63140, Beaverton, OR 77008, USA

This technical report describes possible future directions or extensions to the OpenMP Specification.

The goal of this technical report is to build more widespread existing practice for an expanded OpenMP. It gives advice on extensions or future directions to those vendors who wish to provide them possibly for trial implementation, allows OpenMP to gather early feedback, supports timing and scheduling differences between official OpenMP releases, and offers a preview to users of the future directions of OpenMP with the provisions stated in the next paragraph.

This technical report is non-normative. Some of the components in this technical report may be considered for standardization in a future version of OpenMP, but they are not currently part of any OpenMP specification. Some of the components in this technical report may never be standardized, others may be standardized in a substantially changed form, or it may be standardized as is in its entirety.

OpenMP
Application Programming Interface

Version 5.2 Public Comment Draft, July 2021

This is the Public Comment Draft of the OpenMP API Specification Version 5.2 (15 July 2021) and includes the following internal GitHub issues applied to the 5.1 LaTeX sources, following changes to normalize the file organization: 2153, 2423, 2460, 2510-2511, 2530, 2604, 2607, 2610, 2612, 2614-2616, 2620. 2624-2633, 2635, 2638, 2640, 2642, 2651, 2653-2659, 2661-2663, 2665, 2668-2669, 2679-2680, 2689-2690, 2693, 2698-2699, 2702, 2704-2705, 2713, 2717-2719, 2722-2724, 2726, 2728-2731, 2735, 2738, 2741, 2758-2759, 2761, 2763-2764, 2766, 2770, 2772, 2782

This is a draft; contents will change in official release.

Contents

List of Figures

List of Tables

1 Overview of the OpenMP API

The collection of compiler directives, library routines, and environment variables that this document describes collectively define the specification of the OpenMP Application Program Interface (OpenMP API) for parallelism in C, C++ and Fortran programs.

This specification provides a model for parallel programming that is portable across architectures from different vendors. Compilers from numerous vendors support the OpenMP API. More information about the OpenMP API can be found at the following web site

`http://www.openmp.org`

The directives, library routines, environment variables, and tool support that this document defines allow users to create, to manage, to debug and to analyze parallel programs while permitting portability. The directives extend the C, C++ and Fortran base languages with single program multiple data (SPMD) constructs, tasking constructs, device constructs, worksharing constructs, and synchronization constructs, and they provide support for sharing, mapping and privatizing data. The functionality to control the runtime environment is provided by library routines and environment variables. Compilers that support the OpenMP API often include command line options to enable or to disable interpretation of some or all OpenMP directives.

1.1 Scope

The OpenMP API covers only user-directed parallelization, wherein the programmer explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel. OpenMP-compliant implementations are not required to check for data dependences, data conflicts, race conditions, or deadlocks. Compliant implementations also are not required to check for any code sequences that cause a program to be classified as non-conforming. Application developers are responsible for correctly using the OpenMP API to produce a conforming program. The OpenMP API does not cover compiler-generated automatic parallelization.

1.2 Glossary

1.2.1 Threading Concepts

thread An execution entity with a stack and associated *threadprivate memory*.

OpenMP thread A *thread* that is managed by the OpenMP implementation.

thread number A number that the OpenMP implementation assigns to an OpenMP thread. For threads within the same team, zero identifies the primary thread and consecutive numbers identify the other threads of this team.

idle thread An *OpenMP thread* that is not currently part of any **`parallel`** region.

thread-safe routine A routine that performs the intended function even when executed concurrently (by more than one *thread*).

processor Implementation-defined hardware unit on which one or more *OpenMP threads* can execute.

device An implementation-defined logical execution engine.

> COMMENT: A *device* could have one or more *processors*.

host device The *device* on which the *OpenMP program* begins execution.

target device A device with respect to which the current device performs an operation, as specified by a *device construct* or an OpenMP device memory routine.

parent device For a given **`target`** region, the device on which the corresponding **`target`** construct was encountered.

1.2.2 OpenMP Language Terminology

base language A programming language that serves as the foundation of the OpenMP specification.

> COMMENT: See Section 1.7 for a listing of current *base languages* for the OpenMP API.

base program A program written in a *base language*.

preprocessed code For C/C++, a sequence of preprocessing tokens that result from the first six phases of translation, as defined by the *base language*.

program order An ordering of operations performed by the same *thread* as determined by the execution sequence of operations specified by the *base language*.

> COMMENT: For versions of C and C++ that include base language support for threading, *program order* corresponds to the *sequenced before* relation between operations performed by the same *thread*.

structured block For C/C++, an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom, or an OpenMP *construct*.

For Fortran, a *strictly structured block*, or a *loosely structured block*.

structured block sequence A *structured block*, or, for C/C++, a sequence of two or more executable statements that together have a single entry at the top and a single exit at the bottom.

strictly structured block A single Fortran **`BLOCK`** construct, with a single entry at the top and a single exit at the bottom.

loosely structured block A block of executable constructs, where the first executable construct is not a Fortran **`BLOCK`** construct, with a single entry at the top and a single exit at the bottom, or an OpenMP *construct*.

> COMMENT: In Fortran code, when a *strictly structured block* appears within an OpenMP *construct*, that OpenMP construct does not usually require a paired **`end`** *directive* to define the range of the OpenMP *construct*, while an OpenMP *construct* that contains a *loosely structured block* relies on the paired **`end`** *directive* to define the range of the OpenMP *construct*.

compilation unit For C/C++, a translation unit.

For Fortran, a program unit.

enclosing context For C/C++, the innermost scope enclosing an OpenMP *directive*.

For Fortran, the innermost scoping unit enclosing an OpenMP *directive*.

directive A *base language* mechanism to specify *OpenMP program* behavior.

> COMMENT: See Section 3.1 for a description of OpenMP *directive* syntax in each *base language*.

white space A non-empty sequence of space and/or horizontal tab characters.

OpenMP program A program that consists of a *base program* that is annotated with OpenMP *directives* or that calls OpenMP API runtime library routines.

conforming program An *OpenMP program* that follows all rules and restrictions of the OpenMP specification.

implementation code Implicit code that is introduced by the OpenMP implementation.

metadirective A *directive* that conditionally resolves to another *directive*.

declarative directive An OpenMP *directive* that may only be placed in a declarative context and results in one or more declarations only; it is not associated with the immediate execution of any user code or *implementation code*. For C++, if a declarative directive applies to a function declaration or definition and it is specified with one or more C++ attribute specifiers, the specified attributes must be applied to the function as permitted by the base language. For Fortran, a declarative directive must appear after any **USE**, **IMPORT**, and **IMPLICIT** statements in a declarative context.

executable directive An OpenMP *directive* that appears in an executable context and results in *implementation* code and/or prescribes the manner in which associated user code must execute.

informational directive An OpenMP *directive* that is neither declarative nor executable, but otherwise conveys user code properties to the compiler.

utility directive An OpenMP *directive* that facilitates interactions with the compiler and/or supports code readability; it may be either informational or executable.

stand-alone directive An OpenMP *construct* in which no user code is associated, but may produce implementation code.

construct An OpenMP *executable directive* (and for Fortran, the paired **end** *directive*, if any) and the associated statement, loop nest or *structured block*, if any, not including the code in any called routines. That is, the lexical extent of an *executable directive*.

combined construct A *construct* that is a shortcut for specifying one *construct* immediately nested inside another *construct*. A *combined construct* is semantically identical to that of explicitly specifying the first *construct* containing one instance of the second *construct* and no other statements.

composite construct A *construct* that is composed of two *constructs* but does not have identical semantics to specifying one of the *constructs* immediately nested inside the other. A *composite construct* either adds semantics not included in the *constructs* from which it is composed or provides an effective nesting of the one *construct* inside the other that would otherwise be non-conforming.

constituent construct For a given combined or composite *construct*, a *construct* from which it, or any one of its *constituent constructs*, is composed.

COMMENT: The *constituent constructs* of a **target teams distribute parallel for simd** construct are the following constructs: **target**, **teams distribute parallel for simd**, **teams**, **distribute parallel for simd**, **distribute**, **parallel for simd**, **parallel**, **for simd**, **for**, and **simd**.

leaf construct For a given combined or composite *construct*, a *constituent construct* that is not itself a combined or composite *construct*.

COMMENT: The *leaf constructs* of a **target teams distribute parallel for simd** construct are the following constructs: `target`, `teams`, `distribute`, `parallel`, `for`, and `simd`.

combined target construct A *combined construct* that is composed of a `target` construct along with another construct.

region All code encountered during a specific instance of the execution of a given *construct*, structured block sequence or OpenMP library routine. A *region* includes any code in called routines as well as any *implementation code*. The generation of a *task* at the point where a *task generating construct* is encountered is a part of the *region* of the *encountering thread*. However, an *explicit task region* that corresponds to a *task generating construct* is not part of the *region* of the *encountering thread* unless it is an *included task region*. The point where a `target` or `teams` directive is encountered is a part of the *region* of the *encountering thread*, but the *region* that corresponds to the `target` or `teams` directive is not.

COMMENTS:

A *region* may also be thought of as the dynamic or runtime extent of a *construct* or of an OpenMP library routine.

During the execution of an *OpenMP program*, a *construct* may give rise to many *regions*.

active parallel region A `parallel` *region* that is executed by a *team* consisting of more than one *thread*.

inactive parallel region A `parallel` *region* that is executed by a *team* of only one *thread*.

active target region A `target` *region* that is executed on a *device* other than the *device* that encountered the `target` *construct*.

inactive target region A `target` *region* that is executed on the same *device* that encountered the `target` *construct*.

sequential part All code encountered during the execution of an *initial task region* that is not part of a `parallel` *region* corresponding to a `parallel` *construct* or a `task` *region* corresponding to a `task` *construct*.

COMMENTS:

A *sequential part* is enclosed by an *implicit parallel region*.

Executable statements in called routines may be in both a *sequential part* and any number of explicit `parallel` *regions* at different points in the program execution.

primary thread An *OpenMP thread* that has *thread number* 0. A *primary thread* may be an *initial thread* or the *thread* that encounters a `parallel` *construct*, creates a *team*, generates a set of *implicit tasks*, and then executes one of those *tasks* as *thread* number 0.

worker thread An *OpenMP thread* that is not the *primary thread* of a *team* and that executes one of the *implicit tasks* of a *parallel region*.

parent thread The *thread* that encountered the **parallel** *construct* and generated a **parallel** *region* is the *parent thread* of each of the *threads* in the *team* of that **parallel** *region*. The *primary thread* of a **parallel** *region* is the same *thread* as its *parent thread* with respect to any resources associated with an *OpenMP thread*.

child thread When a thread encounters a **parallel** construct, each of the threads in the generated **parallel** region's team are *child threads* of the encountering *thread*. The **target** or **teams** region's *initial thread* is not a *child thread* of the thread that encountered the **target** or **teams** construct.

ancestor thread For a given *thread*, its *parent thread* or one of its *parent thread's ancestor threads*.

descendent thread For a given *thread*, one of its *child threads* or one of its *child threads' descendent threads*.

team A set of one or more *threads* participating in the execution of a **parallel** *region*.

COMMENTS:

For an *active parallel region*, the team comprises the *primary thread* and at least one additional *thread*.

For an *inactive parallel region*, the *team* comprises only the *primary thread*.

league The set of *teams* created by a **teams** construct.

contention group An initial *thread* and its *descendent threads*.

implicit parallel region An *inactive parallel region* that is not generated from a **parallel** *construct*. *Implicit parallel regions* surround the whole *OpenMP program*, all **target** *regions*, and all **teams** *regions*.

initial thread The *thread* that executes an *implicit parallel region*.

initial team The *team* that comprises an *initial thread* executing an *implicit parallel region*.

nested construct A *construct* (lexically) enclosed by another *construct*.

closely nested construct A *construct* nested inside another *construct* with no other *construct* nested between them.

explicit region A *region* that corresponds to either a *construct* of the same name or a library routine call that explicitly appears in the program.

nested region A *region* (dynamically) enclosed by another *region*. That is, a *region* generated from the execution of another *region* or one of its *nested regions*.

COMMENT: Some nestings are *conforming* and some are not. See Section 17.1 for the restrictions on nesting.

closely nested region A *region nested* inside another *region* with no **`parallel`** *region nested* between them.

strictly nested region A *region nested* inside another *region* with no other *explicit region nested* between them.

all threads All OpenMP *threads* participating in the *OpenMP program*.

current team All *threads* in the *team* executing the innermost enclosing **`parallel`** *region*.

encountering thread For a given *region*, the *thread* that encounters the corresponding *construct*.

all tasks All *tasks* participating in the *OpenMP program*.

current team tasks All *tasks* encountered by the corresponding *team*. The *implicit tasks* constituting the **`parallel`** *region* and any *descendent tasks* encountered during the execution of these *implicit tasks* are included in this set of tasks.

generating task For a given *region*, the task for which execution by a *thread* generated the *region*.

binding thread set The set of *threads* that are affected by, or provide the context for, the execution of a *region*.

The *binding thread set* for a given *region* can be *all threads* on a specified set of devices, *all threads* in a *contention group*, all *primary threads* executing an enclosing **`teams`** *region*, the *current team*, or the *encountering thread*.

COMMENT: The *binding thread set* for a particular *region* is described in its corresponding subsection of this specification.

binding task set The set of *tasks* that are affected by, or provide the context for, the execution of a *region*.

The *binding task* set for a given *region* can be *all tasks*, the *current team tasks*, *all tasks of the current team that are generated in the region*, the *binding implicit task*, or the *generating task*.

COMMENT: The *binding task* set for a particular *region* (if applicable) is described in its corresponding subsection of this specification.

binding region The enclosing *region* that determines the execution context and limits the scope of the effects of the bound *region* is called the *binding region*.

Binding region is not defined for *regions* for which the *binding thread* set is *all threads* or the *encountering thread*, nor is it defined for *regions* for which the *binding task set* is *all tasks*.

orphaned construct A *construct* that gives rise to a *region* for which the *binding thread set* is the *current team*, but is not nested within another *construct* that gives rise to the *binding region*.

work-distribution construct A *construct* that is cooperatively executed by threads in the *binding thread set* of the corresponding region.

worksharing construct A *work-distribution construct* that is executed by the thread team of the innermost enclosing parallel region and includes, by default, an implicit barrier.

device construct An OpenMP *construct* that accepts the **`device`** clause.

cancellable construct An OpenMP *construct* that can be cancelled.

device routine A function (for C/C++ and Fortran) or subroutine (for Fortran) that can be executed on a *target device*, as part of a **`target`** region.

target variant A version of a *device routine* that can only be executed as part of a **`target`** region.

foreign runtime environment A runtime environment that exists outside the OpenMP runtime with which the OpenMP implementation may interoperate.

foreign execution context A context that is instantiated from a *foreign runtime environment* in order to facilitate execution on a given device.

foreign task A unit of work executed in a *foreign execution context*.

indirect device invocation An indirect call to the device version of a procedure on a device other than the host device, through a function pointer (C/C++), a pointer to a member function (C++) or a procedure pointer (Fortran) that refers to the host version of the procedure.

place An unordered set of *processors* on a device.

place list The ordered list that describes all OpenMP *places* available to the execution environment.

place partition An ordered list that corresponds to a contiguous interval in the OpenMP *place list*. It describes the *places* currently available to the execution environment for a given parallel *region*.

place number A number that uniquely identifies a *place* in the *place list*, with zero identifying the first *place* in the *place list*, and each consecutive whole number identifying the next *place* in the *place list*.

thread affinity A binding of *threads* to *places* within the current *place partition*.

SIMD instruction A single machine instruction that can operate on multiple data elements.

SIMD lane A software or hardware mechanism capable of processing one data element from a *SIMD instruction*.

SIMD chunk A set of iterations executed concurrently, each by a *SIMD lane*, by a single *thread* by means of *SIMD instructions*.

memory A storage resource to store and to retrieve variables accessible by OpenMP threads.

memory space A representation of storage resources from which *memory* can be allocated or deallocated. More than one memory space may exist.

memory allocator An OpenMP object that fulfills requests to allocate and to deallocate *memory* for program variables from the storage resources of its associated *memory space*.

handle An opaque reference that uniquely identifies an abstraction.

1.2.3 Loop Terminology

canonical loop nest A loop nest that complies with the rules and restrictions defined in Section 4.4.1.

loop-associated directive An OpenMP *executable directive* for which the associated user code must be a *canonical loop nest*.

associated loop A loop from a *canonical loop nest* that is controlled by a given *loop-associated directive*.

loop nest depth For a *canonical loop nest*, the maximal number of loops, including the outermost loop, that can be associated with a *loop-associated directive*.

logical iteration space For a *loop-associated directive*, the sequence $0,\ldots,N-1$ where N is the number of iterations of the loops associated with the directive. The logical numbering denotes the sequence in which the iterations would be executed if the set of associated loops were executed sequentially.

logical iteration An iteration from the associated loops of a *loop-associated directive*, designated by a logical number from the *logical iteration space* of the associated loops.

logical iteration vector space For a *loop-associated directive* with n associated nested loops, the set of n-tuples $(i_1, \ldots, i_n)$. For the k^{th} associated loop, from outermost to innermost, i_k is its *logical iteration* number as if it was the only associated loop.

logical iteration vector An iteration from the associated nested loops of a *loop-associated directive*, where n is the number of associated loops, designated by an n-tuple from the *logical iteration vector space* of the associated loops.

lexicographic order The total order of two *logical iteration vectors* $\omega_a = (i_1, \ldots, i_n)$ and $\omega_b = (j_1, \ldots, j_n)$, denoted by $\omega_a \leq_{\text{lex}} \omega_b$, where either $\omega_a = \omega_b$ or $\exists m \in \{1, \ldots, n\}$ such that $i_m < j_m$ and $i_k = j_k$ for all $k \in \{1, \ldots, m-1\}$.

product order The partial order of two *logical iteration vectors* $\omega_a = (i_1, \ldots, i_n)$ and $\omega_b = (j_1, \ldots, j_n)$, denoted by $\omega_a \leq_{\text{product}} \omega_b$, where $i_k \leq j_k$ for all $k \in \{1, \ldots, n\}$.

loop transformation construct A construct that is replaced by the loops that result from applying the transformation as defined by its directive to its associated loops.

generated loop A loop that is generated by a *loop transformation construct* and is one of the resulting loops that replace the construct.

SIMD loop A loop that includes at least one *SIMD chunk*.

non-rectangular loop For a loop nest, a loop for which a loop bound references the iteration variable of a surrounding loop in the loop nest.

perfectly nested loop A loop that has no intervening code between it and the body of its surrounding loop. The outermost loop of a loop nest is always perfectly nested.

doacross loop nest A loop nest, consisting of loops that may be associated with the same *loop-associated directive*, that has cross-iteration dependences. An iteration is dependent on one or more lexicographically earlier iterations.

> COMMENT: The `ordered` clause parameter on a worksharing-loop directive identifies the loops associated with the *doacross loop nest*.

1.2.4 Synchronization Terminology

barrier A point in the execution of a program encountered by a *team* of *threads*, beyond which no *thread* in the team may execute until all *threads* in the *team* have reached the barrier and all *explicit tasks* generated by the *team* have executed to completion. If *cancellation* has been requested, threads may proceed to the end of the canceled *region* even if some threads in the team have not reached the *barrier*.

cancellation An action that cancels (that is, aborts) an OpenMP *region* and causes executing *implicit* or *explicit* tasks to proceed to the end of the canceled *region*.

cancellation point A point at which implicit and explicit tasks check if cancellation has been requested. If cancellation has been observed, they perform the *cancellation*.

flush An operation that a *thread* performs to enforce consistency between its view and other *threads*' view of memory.

device-set The set of devices for which a flush operation may enforce memory consistency.

flush property Properties that determine the manner in which a *flush* operation enforces memory consistency. These properties are:

- *strong*: flushes a set of variables from the current thread's temporary view of the memory to the memory;
- *release*: orders memory operations that precede the flush before memory operations performed by a different thread with which it synchronizes;
- *acquire*: orders memory operations that follow the flush after memory operations performed by a different thread that synchronizes with it.

COMMENT: Any *flush* operation has one or more *flush properties*.

strong flush A *flush* operation that has the *strong flush property*.

release flush A *flush* operation that has the *release flush property*.

acquire flush A *flush* operation that has the *acquire flush property*.

atomic operation An operation that is specified by an `atomic` construct or is implicitly performed by the OpenMP implementation and that atomically accesses and/or modifies a specific storage location.

atomic read An *atomic operation* that is specified by an `atomic` construct on which the `read` clause is present.

atomic write An *atomic operation* that is specified by an `atomic` construct on which the `write` clause is present.

atomic update An *atomic operation* that is specified by an `atomic` construct on which the `update` clause is present.

atomic captured update An *atomic update* operation that is specified by an `atomic` construct on which the `capture` clause is present.

atomic conditional update An *atomic update* operation that is specified by an `atomic` construct on which the `compare` clause is present.

read-modify-write An *atomic operation* that reads and writes to a given storage location.

COMMENT: Any *atomic update* is a *read-modify-write* operation.

sequentially consistent atomic construct An `atomic` construct for which the `seq_cst` clause is specified.

non-sequentially consistent atomic construct An `atomic` construct for which the `seq_cst` clause is not specified

sequentially consistent atomic operation An *atomic operation* that is specified by a *sequentially consistent atomic construct*.

1.2.5 Tasking Terminology

task A specific instance of executable code and its data environment that the OpenMP implementation can schedule for execution by threads.

task region A *region* consisting of all code encountered during the execution of a *task*.

implicit task A *task* generated by an *implicit parallel region* or generated when a `parallel` *construct* is encountered during execution.

binding implicit task The *implicit task* of the current thread team assigned to the encountering thread.

explicit task A *task* that is not an *implicit task*.

initial task An *implicit task* associated with an *implicit parallel region*.

current task For a given *thread*, the *task* corresponding to the *task region* in which it is executing.

encountering task For a given *region*, the *current task* of the *encountering thread*.

child task A *task* is a *child task* of its generating *task region*. A *child task region* is not part of its generating *task region*.

sibling tasks *Tasks* that are *child tasks* of the same *task region*.

descendent task A *task* that is the *child task* of a *task region* or of one of its *descendent task regions*.

task completion A condition that is satisfied when a thread reaches the end of the executable code that is associated with the *task* and any *allow-completion* event that is created for the *task* has been fulfilled.

> COMMENT: Completion of the *initial task* that is generated when the program begins occurs at program exit.

task scheduling point A point during the execution of the current *task region* at which it can be suspended to be resumed later; or the point of *task completion*, after which the executing thread may switch to a different *task region*.

task switching The act of a *thread* switching from the execution of one *task* to another *task*.

tied task A *task* that, when its *task region* is suspended, can be resumed only by the same *thread* that was executing it before suspension. That is, the *task* is tied to that *thread*.

untied task A *task* that, when its *task region* is suspended, can be resumed by any *thread* in the team. That is, the *task* is not tied to any *thread*.

undeferred task A *task* for which execution is not deferred with respect to its generating *task region*. That is, its generating *task region* is suspended until execution of the structured block associated with the *undeferred task* is completed.

included task A *task* for which execution is sequentially included in the generating *task region*. That is, an *included task* is *undeferred* and executed by the *encountering thread*.

merged task A *task* for which the *data environment*, inclusive of ICVs, is the same as that of its generating *task region*.

mergeable task A *task* that may be a *merged task* if it is an *undeferred task* or an *included task*.

final task A *task* that forces all of its *child tasks* to become *final* and *included tasks*.

detachable task An *explicit task* that only completes after an associated event variable that represents an *allow-completion* event is fulfilled and execution of the associated structured block has completed.

task dependence An ordering relation between two *sibling tasks*: the *dependent task* and a previously generated *predecessor task*. The *task dependence* is fulfilled when the *predecessor task* has completed.

dependent task A *task* that because of a *task dependence* cannot be executed until its *predecessor tasks* have completed.

mutually exclusive tasks *Tasks* that may be executed in any order, but not at the same time.

predecessor task A *task* that must complete before its *dependent tasks* can be executed.

task synchronization construct A **`taskwait`**, **`taskgroup`**, or a **`barrier`** *construct*.

task generating construct A *construct* that generates one or more *explicit tasks* that are *child tasks* of the *encountering task*.

target task A *mergeable* and *untied task* that is generated by a *device construct* or a call to a device memory routine and that coordinates activity between the current device and the *target device*.

taskgroup set A set of tasks that are logically grouped by a **`taskgroup`** *region*.

1.2.6 Data Terminology

variable A named data storage block, for which the value can be defined and redefined during the execution of a program.

COMMENT: An array element or structure element is a variable that is part of another variable.

scalar variable For C/C++, a scalar variable, as defined by the base language.

For Fortran, a scalar variable with intrinsic type, as defined by the base language, excluding character type.

aggregate variable A variable, such as an array or structure, composed of other variables.

array section A designated subset of the elements of an array that is specified using a subscript notation that can select more than one element.

array item An array, an array section, or an array element.

shape-operator For C/C++, an array shaping operator that reinterprets a pointer expression as an array with one or more specified dimensions.

implicit array For C/C++, the set of array elements of non-array type *T* that may be accessed by applying a sequence of [] operators to a given pointer that is either a pointer to type *T* or a pointer to a multidimensional array of elements of type *T*.

For Fortran, the set of array elements for a given array pointer.

> COMMENT: For C/C++, the implicit array for pointer p with type *T* (*)[10] consists of all accessible elements p[*i*][*j*], for all *i* and *j*=0,1,...,9.

base pointer For C/C++, an lvalue pointer expression that is used by a given lvalue expression or array section to refer indirectly to its storage, where the lvalue expression or array section is part of the implicit array for that lvalue pointer expression.

For Fortran, a data pointer that appears last in the designator for a given variable or array section, where the variable or array section is part of the pointer target for that data pointer.

> COMMENT: For the array section (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers p*i* have a pointer type declaration and identifiers x*i* have an array type declaration, the *base pointer* is: (*p0).x0[k1].p1->p2.

named pointer For C/C++, the *base pointer* of a given lvalue expression or array section, or the *base pointer* of one of its *named pointers*.

For Fortran, the *base pointer* of a given variable or array section, or the *base pointer* of one of its *named pointers*.

> COMMENT: For the array section (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers p*i* have a pointer type declaration and identifiers x*i* have an array type declaration, the *named pointers* are: p0, (*p0).x0[k1].p1, and (*p0).x0[k1].p1->p2.

containing array For C/C++, a non-subscripted array (a *containing array*) to which a series of zero or more array subscript operators and/or . (dot) operators are applied to yield a given lvalue expression or array section for which storage is contained by the array.

For Fortran, an array (a *containing array*) without the **`POINTER`** attribute and without a subscript list to which a series of zero or more array subscript operators and/or component selectors are applied to yield a given variable or array section for which storage is contained by the array.

> COMMENT: An array is a containing array of itself. For the array section (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers p*i* have a pointer type declaration and identifiers x*i* have an array type declaration, the *containing arrays* are: (*p0).x0[k1].p1->p2[k2].x1 and (*p0).x0[k1].p1->p2[k2].x1[k3].x2.

containing structure For C/C++, a structure to which a series of zero or more . (dot) operators and/or array subscript operators are applied to yield a given lvalue expression or array section for which storage is contained by the structure.

For Fortran, a structure to which a series of zero or more component selectors and/or array subscript selectors are applied to yield a given variable or array section for which storage is contained by the structure.

> COMMENT: A structure is a containing structure of itself. For C/C++, a structure pointer *p* to which the -> operator applies is equivalent to the application of a . (dot) operator to (***p*) for the purposes of determining containing sructures.
>
> For the array section (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers p*i* have a pointer type declaration and identifiers x*i* have an array type declaration, the *containing structures* are: *(*p0).x0[k1].p1, (*(*p0).x0[k1].p1).p2[k2] and (*(*p0).x0[k1].p1).p2[k2].x1[k3]

base array For C/C++, a *containing array* of a given lvalue expression or array section that does not appear in the expression of any of its other *containing arrays*.

For Fortran, a *containing array* of a given variable or array section that does not appear in the designator of any of its other *containing arrays*.

> COMMENT: For the array section (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers p*i* have a pointer type declaration and identifiers x*i* have an array type declaration, the *base array* is: (*p0).x0[k1].p1->p2[k2].x1[k3].x2.

named array For C/C++, a *containing array* of a given lvalue expression or array section, or a *containing array* of one of its *named pointers*.

For Fortran, a *containing array* of a given variable or array section, or a *containing array* of one of its *named pointers*.

> COMMENT: For the array section (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers p*i* have a pointer type declaration and identifiers x*i* have an array type declaration, the *named arrays* are: (*p0).x0, (*p0).x0[k1].p1->p2[k2].x1, and (*p0).x0[k1].p1->p2[k2].x1[k3].x2.

base expression The *base array* of a given array section or array element, if it exists; otherwise, the *base pointer* of the array section or array element.

> COMMENT: For the array section (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers p*i* have a

pointer type declaration and identifiers x*i* have an array type declaration, the *base expression* is: (*p0).x0[k1].p1->p2[k2].x1[k3].x2.

More examples for C/C++:

- The *base expression* for x[i] and for x[i:n] is x, if x is an array or pointer.
- The *base expression* for x[5][i] and for x[5][i:n] is x, if x is a pointer to an array or x is 2-dimensional array.
- The *base expression* for y[5][i] and for y[5][i:n] is y[5], if y is an array of pointers or y is a pointer to a pointer.

Examples for Fortran:

- The *base expression* for x(i) and for x(i:j) is x.

base variable For a given data entity that is a variable or array section, a variable denoted by a base language identifier that is either the data entity or is a *containing array* or *containing structure* of the data entity.

COMMENT:

Examples for C/C++:

- The data entities x, x[i], x[:n], x[i].y[j] and x[i].y[:n], where x and y have array type declarations, all have the *base variable* x.
- The lvalue expressions and array sections p[i], p[:n], p[i].y[j] and p[i].y[:n], where p has a pointer type and p[i].y has an array type, has a *base pointer* p but does not have a *base variable*.

Examples for Fortran:

- The data objects x, x(i), x(:n), x(i)%y(j) and x(i)%y(:n), where x and y have array type declarations, all have the *base variable* x.
- The data objects p(i), p(:n), p(i)%y(j) and p(i)%y(:n), where p has a pointer type and p(i)%y has an array type, has a *base pointer* p but does not have a *base variable*.
- For the associated pointer p, p is both its *base variable* and *base pointer*.

attached pointer A pointer variable in a device data environment to which the effect of a **`map`** clause assigns the address of an object, minus some offset, that is created in the device data environment. The pointer is an attached pointer for the remainder of its lifetime in the device data environment.

simply contiguous array section An array section that statically can be determined to have contiguous storage or that, in Fortran, has the **`CONTIGUOUS`** attribute.

structure A structure is a variable that contains one or more variables.

For C/C++: Implemented using struct types.

For C++: Implemented using class types.

For Fortran: Implemented using derived types.

string literal For C/C++, a string literal.

For Fortran, a character literal constant.

private variable With respect to a given set of *task regions* or *SIMD lanes* that bind to the same **`parallel`** *region*, a *variable* for which the name provides access to a different block of storage for each *task region* or *SIMD lane*.

A *variable* that is part of another variable (as an array element or a structure element) cannot be made private independently of other components. If a *variable* is privatized, its components are also private.

shared variable With respect to a given set of *task regions* that bind to the same **`parallel`** *region*, a *variable* for which the name provides access to the same block of storage for each *task region*.

A *variable* that is part of another variable (as an array element or a structure element) cannot be *shared* independently of the other components, except for static data members of C++ classes.

threadprivate variable A *variable* that is replicated, one instance per *thread*, by the OpenMP implementation. Its name then provides access to a different block of storage for each *thread*.

A *variable* that is part of another variable (as an array element or a structure element) cannot be made *threadprivate* independently of the other components, except for static data members of C++ classes. If a *variable* is made *threadprivate*, its components are also *threadprivate*.

threadprivate memory The set of *threadprivate variables* associated with each *thread*.

data environment The *variables* associated with the execution of a given *region*.

device data environment The initial *data environment* associated with a device.

device address An address of an object that may be referenced on a *target device*.

device pointer An *implementation defined handle* that refers to a *device address*.

mapped variable An original *variable* in a *data environment* with a corresponding *variable* in a device *data environment*.

COMMENT: The original and corresponding *variables* may share storage.

mapping operation An operation that establishes or removes a correspondence between a *variable* and another variable in a *device data environment*.

mapper An operation that defines how variables of given type are to be mapped or updated with respect to a device data environment.

user-defined mapper A *mapper* that is defined by a **`declare mapper`** directive.

map-type decay The process that determines the final map types of the map operations that result from mapping a variable with a *user-defined mapper*.

mappable type A type that is valid for a *mapped variable*. If a type is composed from other types (such as the type of an array element or a structure element) and any of the other types are not mappable then the type is not mappable.

COMMENT: Pointer types are *mappable* but the memory block to which the pointer refers is not *mapped*.

For C, the type must be a complete type.

For C++, the type must be a complete type.

In addition, for class types:

- All member functions accessed in any **`target`** region must appear in a declare target directive.

For Fortran, no restrictions on the type except that for derived types:

- All type-bound procedures accessed in any target region must appear in a **`declare target`** directive.

defined For *variables*, the property of having a valid value.

For C, for the contents of *variables*, the property of having a valid value.

For C++, for the contents of *variables* of POD (plain old data) type, the property of having a valid value.

For *variables* of non-POD class type, the property of having been constructed but not subsequently destructed.

For Fortran, for the contents of *variables*, the property of having a valid value. For the allocation or association status of *variables*, the property of having a valid status.

COMMENT: Programs that rely upon *variables* that are not *defined* are *non-conforming programs*.

class type For C++, *variables* declared with one of the **`class`**, **`struct`**, or **`union`** keywords.

static storage duration For C/C++, the lifetime of an object with static storage duration, as defined by the base language.

For Fortran, the lifetime of a variable with a **SAVE** attribute, implicit or explicit, a common block object or a variable declared in a module.

NULL A null pointer. For C, the value **NULL**. For C++, the value **NULL** or the value **nullptr**. For Fortran, the value **C_NULL_PTR**.

non-null value A value that is not *NULL*.

non-null pointer A pointer that is not *NULL*.

1.2.7 Implementation Terminology

supported active levels of parallelism An implementation-defined maximum number of *active parallel regions* that may enclose any region of code in the program.

OpenMP API support Support of at least one active level of parallelism.

nested parallelism support Support of more than one active level of parallelism.

internal control variable A conceptual variable that specifies runtime behavior of a set of *threads* or *tasks* in an *OpenMP program*.

> COMMENT: The acronym ICV is used interchangeably with the term *internal control variable* in the remainder of this specification.

***OpenMP Additional Definitions* document** A document that exists outside of the OpenMP specification and defines additional values that may be used in a *conforming program*. The *OpenMP Additional Definitions document* is available at http://www.openmp.org/.

compliant implementation An implementation of the OpenMP specification that compiles and executes any *conforming program* as defined by the specification.

> COMMENT: A *compliant implementation* may exhibit *unspecified behavior* when compiling or executing a *non-conforming program*.

unspecified behavior A behavior or result that is not specified by the OpenMP specification or not known prior to the compilation or execution of an *OpenMP program*.

Such *unspecified behavior* may result from:

- Issues documented by the OpenMP specification as having *unspecified behavior*.
- A *non-conforming program*.

- A *conforming program* exhibiting an *implementation-defined* behavior.

implementation defined Behavior that must be documented by the implementation, and is allowed to vary among different *compliant implementations*. An implementation is allowed to define this behavior as *unspecified*.

> COMMENT: All features that have *implementation-defined* behavior are documented in Appendix A.

deprecated For a construct, clause, or other feature, the property that it is normative in the current specification but is considered obsolescent and will be removed in the future.

1.2.8 Tool Terminology

tool Code that can observe and/or modify the execution of an application.

first-party tool A tool that executes in the address space of the program that it is monitoring.

third-party tool A tool that executes as a separate process from the process that it is monitoring and potentially controlling.

activated tool A *first-party tool* that successfully completed its initialization.

event A point of interest in the execution of a thread.

native thread A thread defined by an underlying thread implementation.

tool callback A function that a tool provides to an OpenMP implementation to invoke when an associated event occurs.

registering a callback Providing a *tool callback* to an OpenMP implementation.

dispatching a callback at an event Processing a callback when an associated *event* occurs in a manner consistent with the return code provided when a *first-party tool* registered the callback.

thread state An enumeration type that describes the current OpenMP activity of a *thread*. A *thread* can be in only one state at any time.

wait identifier A unique opaque handle associated with each data object (for example, a lock) that the OpenMP runtime uses to enforce mutual exclusion and potentially to cause a thread to wait actively or passively.

frame A storage area on a thread's stack associated with a procedure invocation. A frame includes space for one or more saved registers and often also includes space for saved arguments, local variables, and padding for alignment.

canonical frame address An address associated with a procedure *frame* on a call stack that was the value of the stack pointer immediately prior to calling the procedure for which the frame represents the invocation.

runtime entry point A function interface provided by an OpenMP runtime for use by a tool. A runtime entry point is typically not associated with a global function symbol.

trace record A data structure in which to store information associated with an occurrence of an *event*.

native trace record A *trace record* for an OpenMP device that is in a device-specific format.

signal A software interrupt delivered to a *thread*.

signal handler A function called asynchronously when a *signal* is delivered to a *thread*.

async signal safe The guarantee that interruption by *signal* delivery will not interfere with a set of operations. An async signal safe *runtime entry point* is safe to call from a *signal handler*.

code block A contiguous region of memory that contains code of an OpenMP program to be executed on a device.

OMPT An interface that helps a *first-party tool* monitor the execution of an OpenMP program.

OMPT interface state A state that indicates the permitted interactions between a first-party tool and the OpenMP implementation.

OMPT active An *OMPT interface state* in which the OpenMP implementation is prepared to accept runtime calls from a *first party tool* and will dispatch any registered callbacks and in which a first-party tool can invoke *runtime entry points* if not otherwise restricted.

OMPT pending An *OMPT interface state* in which the OpenMP implementation can only call functions to initialize a *first party tool* and in which a *first-party tool* cannot invoke *runtime entry points*.

OMPT inactive An *OMPT interface state* in which the OpenMP implementation will not make any callbacks and in which a *first-party tool* cannot invoke *runtime entry points*.

OMPD An interface that helps a *third-party tool* inspect the OpenMP state of a program that has begun execution.

OMPD library A dynamically loadable library that implements the *OMPD* interface.

image file An executable or shared library.

address space A collection of logical, virtual, or physical memory address ranges that contain code, stack, and/or data. Address ranges within an address space need not be contiguous. An address space consists of one or more *segments*.

segment A portion of an address space associated with a set of address ranges.

OpenMP architecture The architecture on which an OpenMP *region* executes.

tool architecture The architecture on which an *OMPD* tool executes.

OpenMP process A collection of one or more *threads* and *address spaces*. A process may contain *threads* and *address spaces* for multiple *OpenMP architectures*. At least one thread in an OpenMP process is an OpenMP *thread*. A process may be live or a core file.

address space handle A *handle* that refers to an *address space* within an OpenMP process.

thread handle A *handle* that refers to an OpenMP *thread*.

parallel handle A *handle* that refers to an OpenMP parallel *region*.

task handle A *handle* that refers to an OpenMP task *region*.

descendent handle An output *handle* that is returned from the *OMPD* library in a function that accepts an input *handle*: the output *handle* is a descendent of the input *handle*.

ancestor handle An input *handle* that is passed to the *OMPD* library in a function that returns an output *handle*: the input *handle* is an ancestor of the output *handle*. For a given *handle*, the ancestors of the *handle* are also the ancestors of the handle's descendent.

> COMMENT: A tool cannot use a *handle* in an *OMPD* call if any ancestor of the *handle* has been released, except for *OMPD* calls that release it.

tool context An opaque reference provided by a tool to an *OMPD* library. A *tool context* uniquely identifies an abstraction.

address space context A *tool context* that refers to an *address space* within a process.

thread context A *tool context* that refers to a *native thread*.

native thread identifier An identifier for a *native thread* defined by a thread implementation.

1.3 Execution Model

The OpenMP API uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. The OpenMP API is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library). However, a conforming OpenMP program may execute correctly as a parallel program but not as a sequential program, or may produce different results when executed as a parallel program compared to when it is executed as a sequential program. Further, using different numbers of threads may result in different numeric results because of changes in the association of numeric operations. For example, a serial addition reduction may have a different pattern of addition associations than a parallel reduction. These different associations may change the results of floating-point addition.

An OpenMP program begins as a single thread of execution, called an initial thread. An initial thread executes sequentially, as if the code encountered is part of an implicit task region, called an initial task region, that is generated by the implicit parallel region surrounding the whole program.

The thread that executes the implicit parallel region that surrounds the whole program executes on the *host device*. An implementation may support other devices besides the host device. If supported, these devices are available to the host device for *offloading* code and data. Each device has its own threads that are distinct from threads that execute on another device. Threads cannot migrate from one device to another device. Each device is identified by a device number. The device number for the host device is the value of the total number of non-host devices, while each non-host device has a unique device number that is greater than or equal to zero and less than the device number for the host device. Additionally, the constant **omp_initial_device** can be used as an alias for the host device and the constant **omp_invalid_device** can be used to specify an invalid device number. A *conforming device number* is either a non-negative integer that is less than or equal to **omp_get_num_devices()** or equal to **omp_initial_device** or **omp_invalid_device**.

When a **target** construct is encountered, a new *target task* is generated. The *target task* region encloses the **target** region. The *target task* is complete after the execution of the **target** region is complete.

When a *target task* executes, the enclosed **target** region is executed by an initial thread. The initial thread executes sequentially, as if the target region is part of an initial task region that is generated by an implicit parallel region. The initial thread may execute on the requested *target device*, if it is available and supported. If the target device does not exist or the implementation does not support it, all **target** regions associated with that device execute on the host device.

The implementation must ensure that the **target** region executes as if it were executed in the data environment of the target device unless an **if** clause is present and the **if** clause expression evaluates to *false*.

The **teams** construct creates a *league of teams*, where each team is an initial team that comprises an initial thread that executes the **teams** region. Each initial thread executes sequentially, as if the code encountered is part of an initial task region that is generated by an implicit parallel region associated with each team. Whether the initial threads concurrently execute the **teams** region is unspecified, and a program that relies on their concurrent execution for the purposes of synchronization may deadlock.

If a construct creates a data environment, the data environment is created at the time the construct is encountered. The description of a construct defines whether it creates a data environment.

When any thread encounters a **parallel** construct, the thread creates a team of itself and zero or more additional threads and becomes the primary thread of the new team. A set of implicit tasks, one per thread, is generated. The code for each task is defined by the code inside the **parallel** construct. Each task is assigned to a different thread in the team and becomes tied; that is, it is always executed by the thread to which it is initially assigned. The task region of the task being executed by the encountering thread is suspended, and each member of the new team executes its implicit task. An implicit barrier occurs at the end of the **parallel** region. Only the primary thread resumes execution beyond the end of the **parallel** construct, resuming the task region that was suspended upon encountering the **parallel** construct. Any number of **parallel** constructs can be specified in a single program.

parallel regions may be arbitrarily nested inside each other. If nested parallelism is disabled, or is not supported by the OpenMP implementation, then the new team that is created by a thread that encounters a **parallel** construct inside a **parallel** region will consist only of the encountering thread. However, if nested parallelism is supported and enabled, then the new team can consist of more than one thread. A **parallel** construct may include a **proc_bind** clause to specify the places to use for the threads in the team within the **parallel** region.

When any team encounters a worksharing construct, the work inside the construct is divided among the members of the team, and executed cooperatively instead of being executed by every thread. An implicit barrier occurs at the end of any region that corresponds to a worksharing construct for which the **nowait** clause is not specified. Redundant execution of code by every thread in the team resumes after the end of the worksharing construct.

When any thread encounters a *task generating construct*, one or more explicit tasks are generated. Execution of explicitly generated tasks is assigned to one of the threads in the current team, subject to the thread's availability to execute work. Thus, execution of the new task could be immediate, or deferred until later according to task scheduling constraints and thread availability. Threads are allowed to suspend the current task region at a task scheduling point in order to execute a different task. If the suspended task region is for a tied task, the initially assigned thread later resumes execution of the suspended task region. If the suspended task region is for an untied task, then any thread may resume its execution. Completion of all explicit tasks bound to a given parallel region is guaranteed before the primary thread leaves the implicit barrier at the end of the region. Completion of a subset of all explicit tasks bound to a given parallel region may be specified through the use of task synchronization constructs. Completion of all explicit tasks bound to the implicit parallel region is guaranteed by the time the program exits.

When any thread encounters a **simd** construct, the iterations of the loop associated with the construct may be executed concurrently using the SIMD lanes that are available to the thread.

When a **loop** construct is encountered, the iterations of the loop associated with the construct are executed in the context of its encountering threads, as determined according to its binding region. If the **loop** region binds to a **teams** region, the region is encountered by the set of primary threads that execute the **teams** region. If the **loop** region binds to a **parallel** region, the region is encountered by the team of threads that execute the **parallel** region. Otherwise, the region is encountered by a single thread.

If the `loop` region binds to a `teams` region, the encountering threads may continue execution after the `loop` region without waiting for all iterations to complete; the iterations are guaranteed to complete before the end of the `teams` region. Otherwise, all iterations must complete before the encountering threads continue execution after the `loop` region. All threads that encounter the `loop` construct may participate in the execution of the iterations. Only one of these threads may execute any given iteration.

The `cancel` construct can alter the previously described flow of execution in an OpenMP region. The effect of the `cancel` construct depends on its *construct-type-clause*. If a task encounters a `cancel` construct with a `taskgroup` *construct-type-clause*, then the task activates cancellation and continues execution at the end of its `task` region, which implies completion of that task. Any other task in that `taskgroup` that has begun executing completes execution unless it encounters a `cancellation point` construct, in which case it continues execution at the end of its `task` region, which implies its completion. Other tasks in that `taskgroup` region that have not begun execution are aborted, which implies their completion.

For all other *construct-type-clause* values, if a thread encounters a `cancel` construct, it activates cancellation of the innermost enclosing region of the type specified and the thread continues execution at the end of that region. Threads check if cancellation has been activated for their region at cancellation points and, if so, also resume execution at the end of the canceled region.

If cancellation has been activated, regardless of *construct-type-clause*, threads that are waiting inside a barrier other than an implicit barrier at the end of the canceled region exit the barrier and resume execution at the end of the canceled region. This action can occur before the other threads reach that barrier.

When *runtime error termination* is performed, the effect is as if an `error` directive for which *sev-level* is `fatal` and *action-time* is `execution` is encountered.

Synchronization constructs and library routines are available in the OpenMP API to coordinate tasks and data access in `parallel` regions. In addition, library routines and environment variables are available to control or to query the runtime environment of OpenMP programs.

The OpenMP specification makes no guarantee that input or output to the same file is synchronous when executed in parallel. In this case, the programmer is responsible for synchronizing input and output processing with the assistance of OpenMP synchronization constructs or library routines. For the case where each thread accesses a different file, the programmer does not need to synchronize access.

All concurrency semantics defined by the base language with respect to threads of execution apply to OpenMP threads, unless specified otherwise.

1.4 Memory Model

1.4.1 Structure of the OpenMP Memory Model

The OpenMP API provides a relaxed-consistency, shared-memory model. All OpenMP threads have access to a place to store and to retrieve variables, called the *memory*. A given storage location in the memory may be associated with one or more devices, such that only threads on associated devices have access to it. In addition, each thread is allowed to have its own *temporary view* of the memory. The temporary view of memory for each thread is not a required part of the OpenMP memory model, but can represent any kind of intervening structure, such as machine registers, cache, or other local storage, between the thread and the memory. The temporary view of memory allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable. Each thread also has access to another type of memory that must not be accessed by other threads, called *threadprivate memory*.

A directive that accepts data-sharing attribute clauses determines two kinds of access to variables used in the directive's associated structured block: shared and private. Each variable referenced in the structured block has an original variable, which is the variable by the same name that exists in the program immediately outside the construct. Each reference to a shared variable in the structured block becomes a reference to the original variable. For each private variable referenced in the structured block, a new version of the original variable (of the same type and size) is created in memory for each task or SIMD lane that contains code associated with the directive. Creation of the new version does not alter the value of the original variable. However, the impact of attempts to access the original variable from within the region corresponding to the directive is unspecified; see Section 5.4.3 for additional details. References to a private variable in the structured block refer to the private version of the original variable for the current task or SIMD lane. The relationship between the value of the original variable and the initial or final value of the private version depends on the exact clause that specifies it. Details of this issue, as well as other issues with privatization, are provided in Section 5.

The minimum size at which a memory update may also read and write back adjacent variables that are part of another variable (as array elements or structure elements) is implementation defined but is no larger than the base language requires.

A single access to a variable may be implemented with multiple load or store instructions and, thus, is not guaranteed to be atomic with respect to other accesses to the same variable. Accesses to variables smaller than the implementation defined minimum size or to C or C++ bit-fields may be implemented by reading, modifying, and rewriting a larger unit of memory, and may thus interfere with updates of variables or fields in the same unit of memory.

Two memory operations are considered unordered if the order in which they must complete, as seen by their affected threads, is not specified by the memory consistency guarantees listed in Section 1.4.6. If multiple threads write to the same memory unit (defined consistently with the above access considerations) then a data race occurs if the writes are unordered. Similarly, if at least one thread reads from a memory unit and at least one thread writes to that same memory unit then a data race occurs if the read and write are unordered. If a data race occurs then the result of the program is unspecified.

A private variable in a task region that subsequently generates an inner nested **`parallel`** region is permitted to be made shared for implicit tasks in the inner **`parallel`** region. A private variable in a task region can also be shared by an explicit task region generated during its execution. However, the programmer must use synchronization that ensures that the lifetime of the variable does not end before completion of the explicit task region sharing it. Any other access by one task to the private variables of another task results in unspecified behavior.

A storage location in memory that is associated with a given device has a device address that may be dereferenced by a thread executing on that device, but it may not be generally accessible from other devices. A different device may obtain a device pointer that refers to this device address. The manner in which a program can obtain the referenced device address from a device pointer, outside of mechanisms specified by OpenMP, is implementation defined.

1.4.2 Device Data Environments

When an OpenMP program begins, an implicit **`target data`** region for each device surrounds the whole program. Each device has a device data environment that is defined by its implicit **`target data`** region. Any declare target directives and directives that accept data-mapping attribute clauses determine how an original variable in a data environment is mapped to a corresponding variable in a device data environment.

When an original variable is mapped to a device data environment and a corresponding variable is not present in the device data environment, a new corresponding variable (of the same type and size as the original variable) is created in the device data environment. Conversely, the original variable becomes the new variable's corresponding variable in the device data environment of the device that performs a mapping operation.

The corresponding variable in the device data environment may share storage with the original variable. Writes to the corresponding variable may alter the value of the original variable. The impact of this possibility on memory consistency is discussed in Section 1.4.6. When a task executes in the context of a device data environment, references to the original variable refer to the corresponding variable in the device data environment. If an original variable is not currently mapped and a corresponding variable does not exist in the device data environment then accesses to the original variable result in unspecified behavior unless the **`unified_shared_memory`** clause is specified on a **`requires`** directive for the compilation unit.

The relationship between the value of the original variable and the initial or final value of the corresponding variable depends on the *map-type*. Details of this issue, as well as other issues with mapping a variable, are provided in Section 5.8.2.

The original variable in a data environment and a corresponding variable in a device data environment may share storage. Without intervening synchronization data races can occur.

If a variable has a corresponding variable with which it does not share storage, a write to a storage location designated by the variable causes the value at the corresponding storage location to become undefined.

1.4.3 Memory Management

memory allocator to allocate *memory* in which to store variables. This *memory* will be allocated from the storage resources of the *memory space* associated with the memory allocator. Memory allocators are also used to deallocate previously allocated *memory*. When an OpenMP memory allocator is not used to allocate memory, OpenMP does not prescribe the storage resource for the allocation; the memory for the variables may be allocated in any storage resource.

1.4.4 The Flush Operation

The memory model has relaxed-consistency because a thread's temporary view of memory is not required to be consistent with memory at all times. A value written to a variable can remain in the thread's temporary view until it is forced to memory at a later time. Likewise, a read from a variable may retrieve the value from the thread's temporary view, unless it is forced to read from memory. OpenMP flush operations are used to enforce consistency between a thread's temporary view of memory and memory, or between multiple threads' view of memory.

A flush operation has an associated *device-set* that constrains the threads with which it enforces memory consistency. Consistency is only guaranteed to be enforced between the view of memory of its thread and the view of memory of other threads executing on devices in its device-set. Unless otherwise stated, the device-set of a flush operation only includes the current device.

If a flush operation is a strong flush, it enforces consistency between a thread's temporary view and memory. A strong flush operation is applied to a set of variables called the *flush-set*. A strong flush restricts reordering of memory operations that an implementation might otherwise do. Implementations must not reorder the code for a memory operation for a given variable, or the code for a flush operation for the variable, with respect to a strong flush operation that refers to the same variable.

If a thread has performed a write to its temporary view of a shared variable since its last strong flush of that variable, then when it executes another strong flush of the variable, the strong flush does not complete until the value of the variable has been written to the variable in memory. If a thread performs multiple writes to the same variable between two strong flushes of that variable, the strong flush ensures that the value of the last write is written to the variable in memory. A strong flush of a variable executed by a thread also causes its temporary view of the variable to be discarded, so that if its next memory operation for that variable is a read, then the thread will read from memory and capture the value in its temporary view. When a thread executes a strong flush, no later memory operation by that thread for a variable involved in that strong flush is allowed to start until the strong flush completes. The completion of a strong flush executed by a thread is defined as the point at which all writes to the flush-set performed by the thread before the strong flush are visible in memory to all other threads, and at which that thread's temporary view of the flush-set is discarded.

A strong flush operation provides a guarantee of consistency between a thread's temporary view and memory. Therefore, a strong flush can be used to guarantee that a value written to a variable by one thread may be read by a second thread. To accomplish this, the programmer must ensure that the second thread has not written to the variable since its last strong flush of the variable, and that the following sequence of events are completed in this specific order:

1. The value is written to the variable by the first thread;
2. The variable is flushed, with a strong flush, by the first thread;
3. The variable is flushed, with a strong flush, by the second thread; and
4. The value is read from the variable by the second thread.

If a flush operation is a release flush or acquire flush, it can enforce consistency between the views of memory of two synchronizing threads. A release flush guarantees that any prior operation that writes or reads a shared variable will appear to be completed before any operation that writes or reads the same shared variable and follows an acquire flush with which the release flush synchronizes (see Section 1.4.5 for more details on flush synchronization). A release flush will propagate the values of all shared variables in its temporary view to memory prior to the thread performing any subsequent atomic operation that may establish a synchronization. An acquire flush will discard any value of a shared variable in its temporary view to which the thread has not written since last performing a release flush, and it will load any value of a shared variable propagated by a release flush that synchronizes with it into its temporary view so that it may be subsequently read. Therefore, release and acquire flushes may also be used to guarantee that a value written to a variable by one thread may be read by a second thread. To accomplish this, the programmer must ensure that the second thread has not written to the variable since its last acquire flush, and that the following sequence of events happen in this specific order:

1. The value is written to the variable by the first thread;
2. The first thread performs a release flush;
3. The second thread performs an acquire flush; and
4. The value is read from the variable by the second thread.

Note – OpenMP synchronization operations, described in Section 15 and in Section 18.9, are recommended for enforcing this order. Synchronization through variables is possible but is not recommended because the proper timing of flushes is difficult.

The flush properties that define whether a flush operation is a strong flush, a release flush, or an acquire flush are not mutually disjoint. A flush operation may be a strong flush and a release flush; it may be a strong flush and an acquire flush; it may be a release flush and an acquire flush; or it may be all three.

1.4.5 Flush Synchronization and *Happens Before*

OpenMP supports thread synchronization with the use of release flushes and acquire flushes. For any such synchronization, a release flush is the source of the synchronization and an acquire flush is the sink of the synchronization, such that the release flush *synchronizes with* the acquire flush.

A release flush has one or more associated *release sequences* that define the set of modifications that may be used to establish a synchronization. A release sequence starts with an atomic operation that follows the release flush and modifies a shared variable and additionally includes any read-modify-write atomic operations that read a value taken from some modification in the release sequence. The following rules determine the atomic operation that starts an associated release sequence.

- If a release flush is performed on entry to an atomic operation, that atomic operation starts its release sequence.
- If a release flush is performed in an implicit **flush** region, an atomic operation that is provided by the implementation and that modifies an internal synchronization variable starts its release sequence.
- If a release flush is performed by an explicit **flush** region, any atomic operation that modifies a shared variable and follows the **flush** region in its thread's program order starts an associated release sequence.

An acquire flush is associated with one or more prior atomic operations that read a shared variable and that may be used to establish a synchronization. The following rules determine the associated atomic operation that may establish a synchronization.

- If an acquire flush is performed on exit from an atomic operation, that atomic operation is its associated atomic operation.
- If an acquire flush is performed in an implicit **flush** region, an atomic operation that is provided by the implementation and that reads an internal synchronization variable is its associated atomic operation.

- If an acquire flush is performed by an explicit **flush** region, any atomic operation that reads a shared variable and precedes the **flush** region in its thread's program order is an associated atomic operation.

A release flush synchronizes with an acquire flush if the following conditions are satisfied:

- An atomic operation associated with the acquire flush reads a value written by a modification from a release sequence associated with the release flush; and
- The device on which each flush is performed is in both of their respective device-sets.

An operation *X* *simply happens before* an operation *Y* if any of the following conditions are satisfied:

1. *X* and *Y* are performed by the same thread, and *X* precedes *Y* in the thread's program order;
2. *X* synchronizes with *Y* according to the flush synchronization conditions explained above or according to the base language's definition of *synchronizes with*, if such a definition exists; or
3. Another operation, *Z*, exists such that *X* simply happens before *Z* and *Z* simply happens before *Y*.

An operation *X* *happens before* an operation *Y* if any of the following conditions are satisfied:

1. *X* happens before *Y* according to the base language's definition of *happens before*, if such a definition exists; or
2. *X* simply happens before *Y*.

A variable with an initial value is treated as if the value is stored to the variable by an operation that happens before all operations that access or modify the variable in the program.

1.4.6 OpenMP Memory Consistency

The following rules guarantee an observable completion order for a given pair of memory operations in race-free programs, as seen by all affected threads. If both memory operations are strong flushes, the affected threads are all threads on devices in both of their respective device-sets. If exactly one of the memory operations is a strong flush, the affected threads are all threads on devices in its device-set. Otherwise, the affected threads are all threads.

- If two operations performed by different threads are sequentially consistent atomic operations or they are strong flushes that flush the same variable, then they must be completed as if in some sequential order, seen by all affected threads.
- If two operations performed by the same thread are sequentially consistent atomic operations or they access, modify, or, with a strong flush, flush the same variable, then they must be completed as if in that thread's program order, as seen by all affected threads.
- If two operations are performed by different threads and one happens before the other, then they must be completed as if in that *happens before* order, as seen by all affected threads, if:

- both operations access or modify the same variable;
- both operations are strong flushes that flush the same variable; or
- both operations are sequentially consistent atomic operations.

- Any two atomic memory operations from different **atomic** regions must be completed as if in the same order as the strong flushes implied in their respective regions, as seen by all affected threads.

The flush operation can be specified using the **flush** directive, and is also implied at various locations in an OpenMP program: see Section 15.8.5 for details.

Note – Since flush operations by themselves cannot prevent data races, explicit flush operations are only useful in combination with non-sequentially consistent atomic directives.

OpenMP programs that:

- Do not use non-sequentially consistent atomic directives;
- Do not rely on the accuracy of a *false* result from **omp_test_lock** and **omp_test_nest_lock**; and
- Correctly avoid data races as required in Section 1.4.1,

behave as though operations on shared variables were simply interleaved in an order consistent with the order in which they are performed by each thread. The relaxed consistency model is invisible for such programs, and any explicit flush operations in such programs are redundant.

1.5 Tool Interfaces

The OpenMP API includes two tool interfaces, OMPT and OMPD, to enable development of high-quality, portable, tools that support monitoring, performance, or correctness analysis and debugging of OpenMP programs developed using any implementation of the OpenMP API.

An implementation of the OpenMP API may differ from the abstract execution model described by its specification. The ability of tools that use the OMPT or OMPD interfaces to observe such differences does not constrain implementations of the OpenMP API in any way.

1.5.1 OMPT

first-party tools, provides the following:

- A mechanism to initialize a first-party tool;
- Routines that enable a tool to determine the capabilities of an OpenMP implementation;

- Routines that enable a tool to examine OpenMP state information associated with a thread;
- Mechanisms that enable a tool to map implementation-level calling contexts back to their source-level representations;
- A callback interface that enables a tool to receive notification of OpenMP *events*;
- A tracing interface that enables a tool to trace activity on OpenMP target devices; and
- A runtime library routine that an application can use to control a tool.

OpenMP implementations may differ with respect to the *thread states* that they support, the mutual exclusion implementations that they employ, and the OpenMP events for which tool callbacks are invoked. For some OpenMP events, OpenMP implementations must guarantee that a registered callback will be invoked for each occurrence of the event. For other OpenMP events, OpenMP implementations are permitted to invoke a registered callback for some or no occurrences of the event; for such OpenMP events, however, OpenMP implementations are encouraged to invoke tool callbacks on as many occurrences of the event as is practical. Section 19.2.4 specifies the subset of OMPT callbacks that an OpenMP implementation must support for a minimal implementation of the OMPT interface.

With the exception of the **`omp_control_tool`** runtime library routine for tool control, all other routines in the OMPT interface are intended for use only by tools and are not visible to applications. For that reason, a Fortran binding is provided only for **`omp_control_tool`**; all other OMPT functionality is described with C syntax only.

1.5.2 OMPD

third-party tools, which run as separate processes. An OpenMP implementation must provide an OMPD library that can be dynamically loaded and used by a third-party tool. A third-party tool, such as a debugger, uses the OMPD library to access OpenMP state of a program that has begun execution. OMPD defines the following:

- An interface that an OMPD library exports, which a tool can use to access OpenMP state of a program that has begun execution;
- A callback interface that a tool provides to the OMPD library so that the library can use it to access the OpenMP state of a program that has begun execution; and
- A small number of symbols that must be defined by an OpenMP implementation to help the tool find the correct OMPD library to use for that OpenMP implementation and to facilitate notification of events.

Section 20 describes OMPD in detail.

1.6 OpenMP Compliance

The OpenMP API defines constructs that operate in the context of the base language that is supported by an implementation. If the implementation of the base language does not support a language construct that appears in this document, a compliant OpenMP implementation is not required to support it, with the exception that for Fortran, the implementation must allow case insensitivity for directive and API routines names, and must allow identifiers of more than six characters. An implementation of the OpenMP API is compliant if and only if it compiles and executes all other conforming programs, and supports the tool interfaces, according to the syntax and semantics laid out in Chapters 1 through 20. Appendices A and B as well as sections designated as Notes (see Section 1.8) are for information purposes only and are not part of the specification.

All library, intrinsic and built-in routines provided by the base language must be thread-safe in a compliant implementation. In addition, the implementation of the base language must also be thread-safe. For example, **`ALLOCATE`** and **`DEALLOCATE`** statements must be thread-safe in Fortran. Unsynchronized concurrent use of such routines by different threads must produce correct results (although not necessarily the same as serial execution results, as in the case of random number generation routines).

Starting with Fortran 90, variables with explicit initialization have the **`SAVE`** attribute implicitly. This is not the case in Fortran 77. However, a compliant OpenMP Fortran implementation must give such a variable the **`SAVE`** attribute, regardless of the underlying base language version.

Appendix A lists certain aspects of the OpenMP API that are implementation defined. A compliant implementation must define and document its behavior for each of the items in Appendix A.

1.7 Normative References

- ISO/IEC 9899:1990, *Information Technology - Programming Languages - C*.

  This OpenMP API specification refers to ISO/IEC 9899:1990 as C90.

- ISO/IEC 9899:1999, *Information Technology - Programming Languages - C*.

  This OpenMP API specification refers to ISO/IEC 9899:1999 as C99.

- ISO/IEC 9899:2011, *Information Technology - Programming Languages - C*.

  This OpenMP API specification refers to ISO/IEC 9899:2011 as C11.

- ISO/IEC 9899:2018, *Information Technology - Programming Languages - C*.

  This OpenMP API specification refers to ISO/IEC 9899:2018 as C18.

- ISO/IEC 14882:1998, *Information Technology - Programming Languages - C++*.

  This OpenMP API specification refers to ISO/IEC 14882:1998 as C++98.

- ISO/IEC 14882:2011, *Information Technology - Programming Languages - C++*.

  This OpenMP API specification refers to ISO/IEC 14882:2011 as C++11.
- ISO/IEC 14882:2014, *Information Technology - Programming Languages - C++*.

  This OpenMP API specification refers to ISO/IEC 14882:2014 as C++14.
- ISO/IEC 14882:2017, *Information Technology - Programming Languages - C++*.

  This OpenMP API specification refers to ISO/IEC 14882:2017 as C++17.
- ISO/IEC 14882:2020, *Information Technology - Programming Languages - C++*.

  This OpenMP API specification refers to ISO/IEC 14882:2020 as C++20.
- ISO/IEC 1539:1980, *Information Technology - Programming Languages - Fortran*.

  This OpenMP API specification refers to ISO/IEC 1539:1980 as Fortran 77.
- ISO/IEC 1539:1991, *Information Technology - Programming Languages - Fortran*.

  This OpenMP API specification refers to ISO/IEC 1539:1991 as Fortran 90.
- ISO/IEC 1539-1:1997, *Information Technology - Programming Languages - Fortran*.

  This OpenMP API specification refers to ISO/IEC 1539-1:1997 as Fortran 95.
- ISO/IEC 1539-1:2004, *Information Technology - Programming Languages - Fortran*.

  This OpenMP API specification refers to ISO/IEC 1539-1:2004 as Fortran 2003.
- ISO/IEC 1539-1:2010, *Information Technology - Programming Languages - Fortran*.

  This OpenMP API specification refers to ISO/IEC 1539-1:2010 as Fortran 2008.
- ISO/IEC 1539-1:2018, *Information Technology - Programming Languages - Fortran*.

  This OpenMP API specification refers to ISO/IEC 1539-1:2018 as Fortran 2018. While future versions of the OpenMP specification are expected to address the following features, currently their use may result in unspecified behavior.

  - Declared type of a polymorphic allocatable component in structure constructor
  - **`SELECT RANK`** construct
  - Assumed-rank dummy argument
  - Assumed-type dummy argument
  - Interoperable procedure enhancements

Where this OpenMP API specification refers to C, C++ or Fortran, reference is made to the base language supported by the implementation.

1.8 Organization of this Document

The remainder of this document is structured as normative chapters that define the directives, including their syntax and semantics, the runtime routines and the tool interfaces that comprise the OpenMP API. The document also includes appendices that facilitate maintaining a compliant implementation of the API.

Some sections of this document only apply to programs written in a certain base language. Text that applies only to programs for which the base language is C or C++ is shown as follows:

C / C++

C/C++ specific text...

C / C++

Text that applies only to programs for which the base language is C only is shown as follows:

C

C specific text...

C

Text that applies only to programs for which the base language is C++ only is shown as follows:

C++

C++ specific text...

C++

Text that applies only to programs for which the base language is Fortran is shown as follows:

Fortran

Fortran specific text...

Fortran

Where an entire page consists of base language specific text, a marker is shown at the top of the page. For Fortran-specific text, the marker is:

Fortran (cont.)

For C/C++-specific text, the marker is:

C/C++ (cont.)

Some text is for information only, and is not part of the normative specification. Such text is designated as a note or comment, like this:

Note – Non-normative text...

COMMENT: Non-normative text...

2 Internal Control Variables

An OpenMP implementation must act as if internal control variables (ICVs) control the behavior of an OpenMP program. These ICVs store information such as the number of threads to use for future **parallel** regions, the schedule to use for worksharing loops and whether nested parallelism is enabled or not. The ICVs are given values at various times (described below) during the execution of the program. They are initialized by the implementation itself and may be given values through OpenMP environment variables and through calls to OpenMP API routines. The program can retrieve the values of these ICVs only through OpenMP API routines.

For purposes of exposition, this document refers to the ICVs by certain names, but an implementation is not required to use these names or to offer any way to access the variables other than through the ways shown in Section 2.2.

2.1 ICV Descriptions

The following ICVs store values that affect the operation of **parallel** regions.

- *dyn-var* - controls whether dynamic adjustment of the number of threads is enabled for encountered **parallel** regions. One copy of this ICV exists per data environment.
- *nthreads-var* - controls the number of threads requested for encountered **parallel** regions. One copy of this ICV exists per data environment.
- *thread-limit-var* - controls the maximum number of threads that participate in the contention group. One copy of this ICV exists per data environment.
- *max-active-levels-var* - controls the maximum number of nested active **parallel** regions when the innermost **parallel** region is generated by a given task. One copy of this ICV exists per data environment.
- *place-partition-var* - controls the place partition available to the execution environment for encountered **parallel** regions. One copy of this ICV exists per implicit task.
- *active-levels-var* - the number of nested active **parallel** regions that enclose a given task such that all of the **parallel** regions are enclosed by the outermost initial task region on the device on which the task executes. One copy of this ICV exists per data environment.
- *levels-var* - the number of nested **parallel** regions that enclose a given task such that all of the **parallel** regions are enclosed by the outermost initial task region on the device on which the task executes. One copy of this ICV exists per data environment.

- *bind-var* - controls the binding of OpenMP threads to places. When binding is requested, the variable indicates that the execution environment is advised not to move threads between places. The variable can also provide default thread affinity policies. One copy of this ICV exists per data environment.

The following ICVs store values that affect program execution.

- *run-sched-var* - controls the schedule that is used for worksharing-loop regions when the **`runtime`** schedule kind is specified. One copy of this ICV exists per data environment.
- *stacksize-var* - controls the stack size for threads that the OpenMP implementation creates. One copy of this ICV exists per device.
- *wait-policy-var* - controls the desired behavior of waiting threads. One copy of this ICV exists per device.
- *display-affinity-var* - controls whether to display thread affinity. One copy of this ICV exists for the whole program.
- *affinity-format-var* - controls the thread affinity format when displaying thread affinity. One copy of this ICV exists per device.
- *cancel-var* - controls the desired behavior of the **`cancel`** construct and cancellation points. One copy of this ICV exists for the whole program.
- *default-device-var* - controls the default target device. One copy of this ICV exists per data environment.
- *target-offload-var* - controls the offloading behavior. One copy of this ICV exists for the whole program.
- *max-task-priority-var* - controls the maximum priority value that can be specified in the **`priority`** clause of the **`task`** construct. One copy of this ICV exists for the whole program.

The following ICVs store values that affect the operation of the OMPT tool interface.

- *tool-var* - controls whether an OpenMP implementation will try to register a tool. One copy of this ICV exists for the whole program.
- *tool-libraries-var* - specifies a list of absolute paths to tool libraries for OpenMP devices. One copy of this ICV exists for the whole program.
- *tool-verbose-init-var* - controls whether an OpenMP implementation will verbosely log the registration of a tool. One copy of this ICV exists for the whole program.

The following ICVs store values that affect the operation of the OMPD tool interface.

- *debug-var* - controls whether an OpenMP implementation will collect information that an OMPD library can access to satisfy requests from a tool. One copy of this ICV exists for the whole program.

The following ICVs store values that may be queried by interface routines.

- *num-procs-var* - the number of processors that are available to the device. One copy of this ICV exists per device.
- *thread-num-var* - the thread number of an implicit task within its binding team. One copy of this ICV exists per data environment.
- *final-task-var* - whether a given task is a final task. One copy of this ICV exists per data environment.
- *implicit-task-var* - whether a given task is an implicit task. One copy of this ICV exists per data environment.
- *team-size-var* - the size of the current team. One copy of this ICV exists per data environment.

The following ICV stores values that affect default memory allocation.

- *def-allocator-var* - controls the memory allocator to be used by memory allocation routines, directives and clauses when a memory allocator is not specified by the user. One copy of this ICV exists per implicit task.

The following ICVs store values that affect the operation of **teams** regions.

- *nteams-var* - controls the number of teams requested for encountered **teams** regions. One copy of this ICV exists per device.
- *teams-thread-limit-var* - controls the maximum number of threads that participate in each contention group created by a **teams** construct. One copy of this ICV exists per device.

2.2 ICV Initialization

Table 2.1 shows the ICVs, associated environment variables, and initial values.

TABLE 2.1: ICV Initial Values

ICV	Environment Variable	Initial value
dyn-var	**OMP_DYNAMIC**	See description below
nthreads-var	**OMP_NUM_THREADS**	Implementation defined
run-sched-var	**OMP_SCHEDULE**	Implementation defined
bind-var	**OMP_PROC_BIND**	Implementation defined

table continued on next page

table continued from previous page

ICV	**Environment Variable**	**Initial value**
stacksize-var	**OMP_STACKSIZE**	Implementation defined
wait-policy-var	**OMP_WAIT_POLICY**	Implementation defined
thread-limit-var	**OMP_THREAD_LIMIT**	Implementation defined
max-active-levels-var	**OMP_MAX_ACTIVE_LEVELS**, **OMP_NESTED**, **OMP_NUM_THREADS**, **OMP_PROC_BIND**	Implementation defined
active-levels-var	(none)	*zero*
levels-var	(none)	*zero*
place-partition-var	**OMP_PLACES**	Implementation defined
cancel-var	**OMP_CANCELLATION**	*false*
display-affinity-var	**OMP_DISPLAY_AFFINITY**	*false*
affinity-format-var	**OMP_AFFINITY_FORMAT**	Implementation defined
default-device-var	**OMP_DEFAULT_DEVICE**	See description below
target-offload-var	**OMP_TARGET_OFFLOAD**	**DEFAULT**
max-task-priority-var	**OMP_MAX_TASK_PRIORITY**	*zero*
tool-var	**OMP_TOOL**	*enabled*
tool-libraries-var	**OMP_TOOL_LIBRARIES**	*empty string*
tool-verbose-init-var	**OMP_TOOL_VERBOSE_INIT**	*disabled*
debug-var	**OMP_DEBUG**	*disabled*
num-procs-var	(none)	Implementation defined
thread-num-var	(none)	*zero*
final-task-var	(none)	*false*
implicit-task-var	(none)	*true*
team-size-var	(none)	*one*
def-allocator-var	**OMP_ALLOCATOR**	Implementation defined

table continued on next page

table continued from previous page

ICV	Environment Variable	Initial value
nteams-var	**OMP_NUM_TEAMS**	*zero*
teams-thread-limit-var	**OMP_TEAMS_THREAD_LIMIT**	*zero*

Each ICV that does not have global scope (see Table 2.3) has a set of device-specific environment variables that extend the variables defined in Table 2.1 with the following syntax:

*<ENVIRONMENT VARIABLE>***_DEV***[_<device>]*

where *<ENVIRONMENT VARIABLE>* is one of the variables from Table 2.1 and *<device>* is the device number as specified in the **device** clause (see Section 13).

Semantics

- Each device has its own ICVs.
- The initial value of *dyn-var* is implementation defined if the implementation supports dynamic adjustment of the number of threads; otherwise, the initial value is *false*.
- If *target-offload-var* is **mandatory** and the number of non-host devices is zero then the *default-device-var* is initialized to **omp_invalid_device**. Otherwise, the initial value is an implementation defined non-negative integer that is less than or, if *target-offload-var* is not **mandatory**, equal to **omp_get_initial_device()**.
- The value of the *nthreads-var* ICV is a list.
- The value of the *bind-var* ICV is a list.

The host and non-host device ICVs are initialized before any OpenMP API construct or OpenMP API routine executes. After the initial values are assigned, the values of any OpenMP environment variables that were set by the user are read and the associated ICVs are modified accordingly. If no *<device>* number is specified on the device-specific environment variable then the value is applied to all non-host devices.

Cross References

- **OMP_AFFINITY_FORMAT** environment variable, see Section 21.2.5.
- **OMP_ALLOCATOR** environment variable, see Section 21.5.1.
- **OMP_CANCELLATION** environment variable, see Section 21.2.6.
- **OMP_DEBUG** environment variable, see Section 21.4.1.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 21.2.7.
- **OMP_DISPLAY_AFFINITY** environment variable, see Section 21.2.4.
- **OMP_DYNAMIC** environment variable, see Section 21.1.1.

- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 21.1.4.
- **OMP_MAX_TASK_PRIORITY** environment variable, see Section 21.2.9.
- **OMP_NESTED** environment variable, see Section 21.1.5.
- **OMP_NUM_TEAMS** environment variable, see Section 21.6.1.
- **OMP_NUM_THREADS** environment variable, see Section 21.1.2.
- **OMP_PLACES** environment variable, see Section 21.1.6.
- **OMP_PROC_BIND** environment variable, see Section 21.1.7.
- **OMP_SCHEDULE** environment variable, see Section 21.2.1.
- **OMP_STACKSIZE** environment variable, see Section 21.2.2.
- **OMP_TARGET_OFFLOAD** environment variable, see Section 21.2.8.
- **OMP_TEAMS_THREAD_LIMIT** environment variable, see Section 21.6.2.
- **OMP_THREAD_LIMIT** environment variable, see Section 21.1.3.
- **OMP_TOOL** environment variable, see Section 21.3.1.
- **OMP_TOOL_LIBRARIES** environment variable, see Section 21.3.2.
- **OMP_WAIT_POLICY** environment variable, see Section 21.2.3.

2.3 Modifying and Retrieving ICV Values

Table 2.2 shows the method for modifying and retrieving the values of ICVs through OpenMP API routines. If an ICV is not listed in this table, no OpenMP API routine modifies or retrieves this ICV.

TABLE 2.2: Ways to Modify and to Retrieve ICV Values

ICV	Ways to Modify Value	Ways to Retrieve Value
dyn-var	**omp_set_dynamic**	**omp_get_dynamic**
nthreads-var	**omp_set_num_threads**	**omp_get_max_threads**
run-sched-var	**omp_set_schedule**	**omp_get_schedule**
bind-var	(none)	**omp_get_proc_bind**
thread-limit-var	**target** construct, **teams** construct	**omp_get_thread_limit**
max-active-levels-var	**omp_set_max_active_levels**, **omp_set_nested**	**omp_get_max_active_levels**

table continued on next page

table continued from previous page

ICV	Ways to Modify Value	Ways to Retrieve Value
active-levels-var	(none)	**omp_get_active_level**
levels-var	(none)	**omp_get_level**
place-partition-var	(none)	See description below
cancel-var	(none)	**omp_get_cancellation**
affinity-format-var	**omp_set_affinity_format**	**omp_get_affinity_format**
default-device-var	**omp_set_default_device**	**omp_get_default_device**
max-task-priority-var	(none)	**omp_get_max_task_priority**
num-procs-var	(none)	**omp_get_num_procs**
thread-num-var	(none)	**omp_get_thread_num**
final-task-var	(none)	**omp_in_final**
team-size-var	(none)	**omp_get_num_threads**
def-allocator-var	**omp_set_default_allocator**	**omp_get_default_allocator**
nteams-var	**omp_set_num_teams**	**omp_get_max_teams**
teams-thread-limit-var	**omp_set_teams_thread_limit**	**omp_get_teams_thread_limit**

Semantics

- The value of the *nthreads-var* ICV is a list. The runtime call **omp_set_num_threads** sets the value of the first element of this list, and **omp_get_max_threads** retrieves the value of the first element of this list.
- The value of the *bind-var* ICV is a list. The runtime call **omp_get_proc_bind** retrieves the value of the first element of this list.
- Detailed values in the *place-partition-var* ICV are retrieved using the runtime calls **omp_get_partition_num_places**, **omp_get_partition_place_nums**, **omp_get_place_num_procs**, and **omp_get_place_proc_ids**.

Cross References

- **thread_limit** clause of the **teams** construct, see Section 10.2.
- **omp_get_active_level** routine, see Section 18.2.20.
- **omp_get_affinity_format** routine, see Section 18.3.9.
- **omp_get_cancellation** routine, see Section 18.2.8.
- **omp_get_default_allocator** routine, see Section 18.13.5.

- **omp_get_default_device** routine, see Section 18.7.3.
- **omp_get_dynamic** routine, see Section 18.2.7.
- **omp_get_level** routine, see Section 18.2.17.
- **omp_get_max_active_levels** routine, see Section 18.2.16.
- **omp_get_max_task_priority** routine, see Section 18.5.1.
- **omp_get_max_teams** routine, see Section 18.4.4.
- **omp_get_max_threads** routine, see Section 18.2.3.
- **omp_get_num_procs** routine, see Section 18.7.1.
- **omp_get_num_threads** routine, see Section 18.2.2.
- **omp_get_partition_num_places** routine, see Section 18.3.6.
- **omp_get_partition_place_nums** routine, see Section 18.3.7.
- **omp_get_place_num_procs** routine, see Section 18.3.3.
- **omp_get_place_proc_ids** routine, see Section 18.3.4.
- **omp_get_proc_bind** routine, see Section 18.3.1.
- **omp_get_schedule** routine, see Section 18.2.12.
- **omp_get_supported_active_levels**, see Section 18.2.14.
- **omp_get_teams_thread_limit** routine, see Section 18.4.6.
- **omp_get_thread_limit** routine, see Section 18.2.13.
- **omp_get_thread_num** routine, see Section 18.2.4.
- **omp_in_final** routine, see Section 18.5.2.
- **omp_set_affinity_format** routine, see Section 18.3.8.
- **omp_set_default_allocator** routine, see Section 18.13.4.
- **omp_set_default_device** routine, see Section 18.7.2.
- **omp_set_dynamic** routine, see Section 18.2.6.
- **omp_set_max_active_levels** routine, see Section 18.2.15.
- **omp_set_nested** routine, see Section 18.2.9.
- **omp_set_num_teams** routine, see Section 18.4.3.
- **omp_set_num_threads** routine, see Section 18.2.1.
- **omp_set_schedule** routine, see Section 18.2.11.

- **omp_set_teams_thread_limit** routine, see Section 18.4.5.
- **thread_limit** clause of the **target** construct, see Section 13.8.

2.4 How ICVs are Scoped

Table 2.3 shows the ICVs and their scope.

TABLE 2.3: Scopes of ICVs

ICV	Scope
dyn-var	data environment
nthreads-var	data environment
run-sched-var	data environment
bind-var	data environment
stacksize-var	device
wait-policy-var	device
thread-limit-var	data environment
max-active-levels-var	data environment
active-levels-var	data environment
levels-var	data environment
place-partition-var	implicit task
cancel-var	global
display-affinity-var	global
affinity-format-var	device
default-device-var	data environment
target-offload-var	global
max-task-priority-var	global
tool-var	global
tool-libraries-var	global
tool-verbose-init-var	global
debug-var	global
num-procs-var	device
thread-num-var	implicit task
final-task-var	data environment
implicit-task-var	data environment
team-size-var	team
def-allocator-var	implicit task
nteams-var	device
teams-thread-limit-var	device

Semantics

- One copy of each ICV with device scope exists per device.
- Each data environment has its own copies of ICVs with data environment scope.
- Each implicit task has its own copy of ICVs with implicit task scope.

Calls to OpenMP API routines retrieve or modify data environment scoped ICVs in the data environment of their binding tasks.

2.4.1 How the Per-Data Environment ICVs Work

When a **`task`** construct, a **`parallel`** construct or a **`teams`** construct is encountered, each generated task inherits the values of the data environment scoped ICVs from each generating task's ICV values.

When a **`parallel`** construct is encountered, the value of each ICV with implicit task scope is inherited from the implicit binding task of the generating task unless otherwise specified.

When a **`task`** construct is encountered, the generated task inherits the value of *nthreads-var* from the generating task's *nthreads-var* value. When a **`parallel`** construct is encountered, and the generating task's *nthreads-var* list contains a single element, the generated implicit tasks inherit that list as the value of *nthreads-var*. When a **`parallel`** construct is encountered, and the generating task's *nthreads-var* list contains multiple elements, the generated implicit tasks inherit the value of *nthreads-var* as the list obtained by deletion of the first element from the generating task's *nthreads-var* value. The *bind-var* ICV is handled in the same way as the *nthreads-var* ICV.

When a *target task* executes an active **`target`** region, the generated initial task uses the values of the data environment scoped ICVs from the device data environment ICV values of the device that will execute the region.

When a *target task* executes an inactive **`target`** region, the generated initial task uses the values of the data environment scoped ICVs from the data environment of the task that encountered the **`target`** construct.

If a **`target`** construct with a **`thread_limit`** clause is encountered, the *thread-limit-var* ICV from the data environment of the generated initial task is instead set to an implementation defined value between one and the value specified in the clause.

If a **`target`** construct with no **`thread_limit`** clause is encountered, the *thread-limit-var* ICV from the data environment of the generated initial task is set to an implementation defined value that is greater than zero.

If a **`teams`** construct with a **`thread_limit`** clause is encountered, the *thread-limit-var* ICV from the data environment of the initial task for each team is instead set to an implementation defined value between one and the value specified in the clause.

If a **teams** construct with no **thread_limit** clause is encountered, the *thread-limit-var* ICV from the data environment of the initial task of each team is set to an implementation defined value that is greater than zero and does not exceed *teams-thread-limit-var*, if *teams-thread-limit-var* is greater than zero.

When encountering a worksharing-loop region for which the **runtime** schedule kind is specified, all implicit task regions that constitute the binding parallel region must have the same value for *run-sched-var* in their data environments. Otherwise, the behavior is unspecified.

2.5 ICV Override Relationships

Table 2.4 shows the override relationships among construct clauses and ICVs. The table only lists ICVs that can be overwritten by a clause.

TABLE 2.4: ICV Override Relationships

ICV	construct clause, if used
nthreads-var	**num_threads**
run-sched-var	**schedule**
bind-var	**proc_bind**
def-allocator-var	**allocate**
nteams-var	**num_teams**
teams-thread-limit-var	**thread_limit**

Semantics

- The **num_threads** clause overrides the value of the first element of the *nthreads-var* ICV.
- If a **schedule** clause specifies a modifier then that modifier overrides any modifier that is specified in the *run-sched-var* ICV.
- If *bind-var* is not set to *false* then the **proc_bind** clause overrides the value of the first element of the *bind-var* ICV; otherwise, the **proc_bind** clause has no effect.

Cross References

- **allocate** clause, Section 6.7.
- **allocate** directivee, Section 6.6.
- **proc_bind** clause, Section 10.1.
- **thread_limit** clause, see Section 10.2.

- **num_threads** clause, see Section 10.1.1.
- Worksharing-loop construct, see Section 11.5.
- **schedule** clause, see Section 11.5.3.

3 Directive and Construct Syntax

This chapter describes the syntax of OpenMP directives, clauses and any related base language code. OpenMP directives are specified with various base-language mechanisms that allow compilers to ignore OpenMP directives and conditionally compiled code if support of the OpenMP API is not provided or enabled. A compliant implementation must provide an option or interface that ensures that underlying support of all OpenMP directives and OpenMP conditional compilation mechanisms is enabled. In the remainder of this document, the phrase *OpenMP compilation* is used to mean a compilation with these OpenMP features enabled.

Restrictions

The following restrictions apply to OpenMP directives:

- A program must not depend on any ordering of the evaluations of the expressions that appear in the clauses specified on a directive, unless otherwise specified.
- A program must not depend on any side effects of the evaluations of the expressions that appear in the clauses specified on a directive, unless otherwise specified.

C

- A declarative directive may not be used in place of a substatement in a selection statement, in place of the loop body in an iteration statement, or in place of the statement that follows a label.

C

C++

- A declarative directive may not be used in place of a substatement in a selection statement or iteration statement, or in place of the statement that follows a label.

C++

Fortran

- OpenMP directives, except **simd** and declarative directives, may not appear in pure procedures.
- OpenMP directives may not appear in the **WHERE**, **FORALL** or **DO CONCURRENT** constructs.

Fortran

3.1 Directive Format

This section defines several categories of directives and constructs. OpenMP directives are specified with a *directive-specification*. A *directive-specification* consists of the *directive-specifier* and any clauses that may optionally be associated with the OpenMP directive:

directive-specifier [[**,** *] clause[[* **,** *] clause] ...]*

The *directive-specifier* is:

directive-name

or for argument-modified directives:

directive-name [**(***directive-arguments***)** *]*

C / C++

White space in a *directive-name* is not optional.

C / C++

Some OpenMP directives specify a paired **end** directive, where the *directive-name* of the paired **end** directives is:

- If *directive-name* starts with **begin**, the *end-directive-name* replaces **begin** with **end**
- otherwise it is **end** *directive-name* unless otherwise specified.

The *directive-specification* of a paired **end** directive may include one or more optional *end-clause*:

directive-specifier [end-clause[[**,** *] end-clause]...]*

where *end-clause* has the *end-clause* property, which explicitly allows it on a paired **end** directive.

C / C++

An OpenMP *directive* may be specified as a pragma directive:

#pragma omp *directive-specification new-line*

or:

_Pragma("omp *directive-specification***")**

The use of **omp** as the first preprocessing token of a pragma directive is reserved for OpenMP directives that are defined in this specification. The use of **ompx** as the first preprocessing token of a pragma directive is reserved for implementation-defined extensions to the OpenMP directives.

Note – In the following example OpenMP *directive*, **depobj** is the *directive-name*, **o** is the *directive-arguments*. **depobj(o)** is the *directive-specifier* and **depobj(o) depend(inout: d)** is the *directive-specification*,

```
#pragma omp depobj(o) depend(inout: d)
```

White space can be used before and after the **#**. Preprocessing tokens following **#pragma omp** are subject to macro replacement.

C / C++

C++

In C++11 and higher, an OpenMP *directive* may be specified as a C++ attribute specifier:

```
[[ omp :: directive-attr ]]
```

or

```
[[ using omp : directive-attr ]]
```

where *directive-attr* is

```
directive( directive-specification )
```

or

```
sequence( [omp::]directive-attr [, [omp::]directive-attr]... )
```

Multiple attributes on the same statement are allowed. Attribute directives that apply to the same statement are unordered unless the **sequence** attribute is specified, in which case the right-to-left ordering applies. The **omp::** namespace qualifier within a **sequence** attribute is optional. The application of multiple attributes in a **sequence** attribute is ordered as if each directive had been specified as a pragma directive on subsequent lines.

Note – This example shows the expected transformation:

```
[[ omp::sequence(directive(parallel), directive(for)) ]]
for(...) {}
// becomes
#pragma omp parallel
#pragma omp for
for(...) {}
```

The use of **omp** as the attribute namespace of an attribute specifier, or as the optional namespace qualifier within a **sequence** attribute, is reserved for OpenMP directives that are defined in this specification. The use of **ompx** as the attribute namespace of an attribute specifier, or as the optional namespace qualifier within a **`sequence`** attribute, is reserved for implementation-defined extensions to the OpenMP directives.

The pragma and attribute forms are interchangeable for any OpenMP *directive*. Some OpenMP directives may be composed of consecutive attribute specifiers if specified in their syntax. Any two consecutive attribute specifiers may be reordered or expressed as a single attribute specifier, as permitted by the base language, without changing the behavior of the OpenMP directive.

C++

C / C++

Directives are case-sensitive. Each expression used in the OpenMP syntax inside of a clause must be a valid *assignment-expression* of the base language unless otherwise specified.

C / C++

C++

Directives may not appear in **`constexpr`** functions or in constant expressions.

C++

Fortran

An OpenMP *directive* for Fortran is specified with a stylized comment as follows:

sentinel directive-specification

All OpenMP compiler directives must begin with a directive *sentinel*. The format of a sentinel differs between fixed form and free form source files, as described in Section 3.1.1 and Section 3.1.2. In order to simplify the presentation, free form is used for the syntax of OpenMP directives for Fortran throughout this document, except as noted.

Directives are case insensitive. Directives cannot be embedded within continued statements, and statements cannot be embedded within directives. Each expression used in the OpenMP syntax inside of a clause must be a valid *expression* of the base language unless otherwise specified.

Fortran

A directive may be categorized as one of the following:

- meta
- declarative
- executable
- informational
- utility
- subsidiary

Base language code can be associated with directives. The directive's association can be categorized as:

- none

- block-associated
- loop-associated
- declaration-associated
- delimited
- separating

A *directive* and its associated base language code constitute a syntactic formation that follows the syntax given below. The *end-directive* in a specified formation refers to the paired **end** directive for the *directive*. An OpenMP construct is a formation for which the *directive* is executable.

Directives with an association of none are not associated with any base language code. The resulting formation therefore has the following syntax:

```
directive
```

Formations that result from a block-associated directive have the following syntax:

C / C++

```
directive
    structured-block
```

C / C++

Fortran

```
directive
    structured-block
[end-directive]
```

If *structured-block* is a loosely structured block, *end-directive* is required. If *structured-block* is a strictly structured block, *end-directive* is optional.

Fortran

Loop-associated directives are block-associated directives for which the associated *structured-block* is a *loop-nest*, a canonical loop nest.

Fortran

For a loop-associated directive, the paired **end** directive is optional.

Fortran

C / C++

Formations that result from a declaration-associated directive have the following syntax:

declaration-associated-specification

where *declaration-associated-specification* is either:

directive
    function-definition-or-declaration

or:

directive
declaration-associated-specification

In all cases the *directive* is associated with the *function-definition-or-declaration*.

C / C++

Fortran

The formation that results from a declaration-associated directive in Fortran has the same syntax as the formation for a directive with an association of none.

If a directive appears in the specification part of a module then the behavior is as if that directive appears after any references to that module.

Fortran

The formation that results from a delimited directive has the following syntax:

directive
    base-language-code
end-directive

Separating directives may be used to separate a *structured-block* into multiple *structured-block-sequence*s.

Separating directives and the body of the containing structured block have the following syntax:

structured-block-sequence
directive
structured-block-sequence
[directive
structured-block-sequence ...]

Restrictions

Restrictions to directive format are as follows:

C / C++

- A directive that uses the attribute syntax cannot be applied to the same statement or associated declaration as a directive that uses the pragma syntax.
- For any directive that has a paired **end** directive, both directives must use either the attribute syntax or the pragma syntax.

C / C++

- Orphaned separating directives are prohibited. That is, the separating directives must appear within the structured block associated with the same construct with which it is associated and must not be encountered elsewhere in the region of that associated construct.

Restrictions on explicit OpenMP regions (that arise from executable directives) are as follows:

C++

- A **throw** executed inside a region that arises from a thread-limiting directive must cause execution to resume within the same region, and the same thread that threw the exception must catch it. If the directive is also exception-aborting then whether the exception is caught or the **throw** is instead treated as an **error** directive for which *sev-level* is **fatal** and *action-time* is **execution** is implementation defined.

C++

Fortran

- If more than one image is executing the program, any image control statement, **ERROR STOP** statement, **FAIL IMAGE** statement, collective subroutine call or access to a coindexed object that appears in an explicit OpenMP region will result in unspecified behavior.

Fortran

Restrictions to stand-alone directives are as follows:

- A stand-alone directive may be placed only at a point where a base language executable statement is allowed.

C

- A stand-alone directive may not be used in place of a substatement in a selection statement, in place of the loop body in an iteration statement, or in place of the statement that follows a label.

C

C++

- A stand-alone directive may not be used in place of a substatement in a selection statement or iteration statement, or in place of the statement that follows a label.

C++

Fortran

3.1.1 Fixed Source Form Directives

The following sentinels are recognized in fixed form source files:

```
!$omp | c$omp | *$omp | !$omx | c$omx | *$omx
```

The sentinels that end with **omp** are reserved for OpenMP directives that are defined in this specification. The sentinels that end with **omx** are reserved for implementation-defined extensions to the OpenMP directives.

Sentinels must start in column 1 and appear as a single word with no intervening characters. Fortran fixed form line length, white space, continuation, and column rules apply to the directive line. Initial directive lines must have a space or a zero in column 6, and continuation directive lines must have a character other than a space or a zero in column 6.

Comments may appear on the same line as a directive. The exclamation point initiates a comment when it appears after column 6. The comment extends to the end of the source line and is ignored. If the first non-blank character after the directive sentinel of an initial or continuation directive line is an exclamation point, the line is ignored.

Note – In the following example, the three formats for specifying the directive are equivalent (the first line represents the position of the first 9 columns):

```
c23456789
!$omp parallel do shared(a,b,c)

c$omp parallel do
c$omp+shared(a,b,c)

c$omp paralleldoshared(a,b,c)
```

Fortran

3.1.2 Free Source Form Directives

The following sentinels are recognized in free form source files:

```
!$omp | !$ompx
```

The **!$omp** sentinel is reserved for OpenMP directives that are defined in this specification. The **!$ompx** sentinel is reserved for implementation-defined extensions to the OpenMP directives.

The sentinel can appear in any column as long as it is preceded only by white space. It must appear as a single word with no intervening white space. Fortran free form line length and white space rules apply to the directive line. Initial directive lines must have a space after the sentinel. Fortran free form continuation rules apply. Thus, continued directive lines must have an ampersand (**&**) as the last non-blank character on the line, prior to any comment placed inside the directive; continuation directive lines can have an ampersand after the directive sentinel with optional white space before and after the ampersand.

Comments may appear on the same line as a directive. The exclamation point (**!**) initiates a comment. The comment extends to the end of the source line and is ignored. If the first non-blank character after the directive sentinel is an exclamation point, the line is ignored.

One or more blanks or horizontal tabs are optional to separate adjacent keywords in *directive-names* unless otherwise specified.

Note – In the following example the three formats for specifying the directive are equivalent (the first line represents the position of the first 9 columns):

```
!23456789
       !$omp parallel do &
                 !$omp shared(a,b,c)

       !$omp parallel &
      !$omp&do shared(a,b,c)

!$omp paralleldo shared(a,b,c)
```

3.2 Clause Format

This section defines the format and categories of OpenMP clauses. OpenMP clauses are specified as part of a *directive-specification*. Clauses are optional and, thus, may be omitted from a *directive-specification* unless otherwise specified. The order in which clauses appear on directives is not significant unless otherwise specified. A *clause-specification* specifies each OpenMP clause in a *directive-specification* where *clause-specification* for inarguable clauses is simply:

clause-name

Inarguable clauses often form natural groupings that have similar semantic effect and so are frequently specified as a clause grouping. For argument-modified clauses, *clause-specification* is:

clause-name **(***clause-argument-specification [***;** *clause-argument-specification [***;** *...]]***)**

C / C++

White space in a *clause-name* is prohibited. White space within a *clause-argument-specification* and between another *clause-argument-specification* is optional.

C / C++

An implementation may allow clauses with clause names that start with the **`ompx_`** prefix for use on any OpenMP directive, and the format and semantics of any such clause is implementation defined. All other clause names are reserved.

For argument-modified clauses, the first *clause-argument-specification* is required unless otherwise explicitly stated while additional ones are only permitted on clauses that explicitly allow them. When the first one is omitted, the syntax is identical to an inarguable clause. Clause arguments may be unmodified or modified. For an unmodified argument, *clause-argument-specification* is:

clause-argument-list

Unless otherwise specified, modified arguments are pre-modified, for which the format is:

[modifier-specification [[`,` *modifier-specification]*`,`*...]* `:` *] clause-argument-list*

A few modified arguments are explicitly specified as post-modified, for which the format is:

clause-argument-list[`:` *[modifier-specification [[*`,` *modifier-specification]* `,` *...]]]*

For many OpenMP clauses, *clause-argument-list* is an OpenMP argument list, which is a comma-separated list of a specific kind of list items (see Section 3.2.1), in which case the format of *clause-argument-list* is:

argument-name

For all other OpenMP clauses, *clause-argument-list* is a comma-separated list of arguments so the format is:

argument-name [`,` *argument-name [*`,`*...]]*

In most of these cases, the list only has a single item so the format of *clause-argument-list* is again:

argument-name

In all cases, white space in *clause-argument-list* is optional.

Clause argument modifiers may be simple or complex. Almost all clause arguments are simple, for which the format of *modifier-specification* is:

modifier-name

The format of a complex modifier is:

modifier-name **(***modifier-parameter-specification***)** where *modifier-paramater-specification* is a comma-separated list of arguments as defined above for *clause-argument-list*. The position of each *modifier-argument-name* in the list is significant.

Each *argument-name* and *modifier-name* is an OpenMP term that may be used in the definitions of the clause and any directives on which the clause may appear. Syntactically, each of these terms is one of the following:

- *keyword*: An OpenMP keyword
- *OpenMP identifier*: An OpenMP identifier
- *OpenMP argument list*: An OpenMP argument list
- *expression*: An expression of some OpenMP type
- *OpenMP stylized expression*: An OpenMP stylized expression

A particular lexical instantiation of an argument specifies a parameter of the clause, while a lexical instantiation of a modifier and its parameters affects how or when the argument is applied.

The order of arguments must match the order in the *clause-specification*. The order of modifiers in a *clause-argument-specification* is not significant unless otherwise specified.

Each clause has properties that govern its use on a directive that accepts it as defined in the restrictions listed in this section or in the section that defines the clause or the directive. Similarly, arguments and modifiers that are defined in a clause syntax have properties that govern their use. These general clause, argument and modifier properties are defined as:

- optional
- required
- unique
- repeatable
- ultimate
- constant
- positive
- non-negative
- region-invariant

Some of the properties form subsets. If a clause, argument or modifier is optional then it is not required. If a clause, argument or modifier is unique then it is not repeatable. Clauses are optional and repeatable unless otherwise specified. A *clause-specification* can omit optional arguments and modifiers. Each argument is required and unique unless otherwise specified. Each modifier is optional and unique unless otherwise specified. If all arguments and modifiers of an argument-modified clause are optional then the parentheses of the syntax are also optional.

Note – In the following example, **`depend(inout: d)`** is a *clause-specification*, **`depend`** is the *clause-name* and **`inout: d`** is a *clause-argument-specification*. The **`depend`** clause has an argument with the *argument-name locator-list*, which syntactically is the OpenMP locator list **`d`** in the example. Similarly, the **`depend`** clause accepts a simple clause modifier with the name *takes-dependence-type*. Syntactically, *task-dependence-type* is the keyword **`inout`** in the example.

```
#pragma omp depobj(o) depend(inout: d)
```

The clauses that a directive accepts may form sets. These sets may imply restrictions on their use on that directive or may otherwise capture properties for the clauses on the directive. While specific properties may be defined for a clause set on a particular directive, the following clause-set properties have general meanings and implications:

- optional
- required
- unique
- exclusive
- fully exclusive

All clauses that are specified as a clause grouping form a clause set for which properties are specified with the specification of the grouping. Some directives accept a a clause grouping for which each member is a *directive-name* of a directive that has a specific property. These groupings are required, unique and fully exclusive unless otherwise specified.

Restrictions

Restrictions to clauses and clause sets are as follows:

- A required clause for a directive must appear on the directive.
- A unique clause for a directive may appear at most once on the directive.
- If a clause is a member of a set that has the unique property for a directive then the clause has the unique property for that directive regardless of whether it has the unique property when it is not part of such a set.
- If one clause of an exclusive set appears on a directive, no other clauses with a different *clause-name* in that set may appear on the directive.
- At most one clause of a fully exclusive set may appear on a directive.
- A required argument must appear in the *clause-specification*.
- A unique argument may appear at most once in a *clause-specification*.

- A required modifier must appear in the *clause-argument-specification*.
- A unique modifier may appear at most once in a *clause-argument-specification*.
- If a clause is pre-modified, an ultimate modifier must be the last modifier in a *clause-argument-specification* in which any modifier appears.
- If a clause is post-modified, an ultimate modifier must be the first modifier in a *clause-argument-specification* in which any modifier appears.
- A constant argument or parameter must be a compile-time constant.
- A positive argument or parameter must be greater than zero; a non-negative argument or parameter must be greater than or equal to zero.
- A region-invariant argument or parameter must have the same value throughout any given execution of the construct or, for declarative directives, execution of the function or subroutine with which the declaration is associated.

Cross References

- Directive format, see Section 3.1.
- OpenMP argument lists, see Section 3.2.1.
- OpenMP stylized expressions, see Section 4.2.
- OpenMP types and identifiers, see Section 4.1.

3.2.1 OpenMP Argument Lists

OpenMP defines several kinds of lists, each of which can be used as syntactic instances of clause arguments. A list of any OpenMP type consists of a comma-separated collection of expressions of that OpenMP type. A variable list consists of a comma-separated collection of one or more *variable list items*. An extended list consists of a comma-separated collection of one or more *extended list items*. A locator list consists of a comma-separated collection of one or more *locator list items*. A parameter list consists of a comma-separated collection of one or more *parameter list items*. A type-name list consists of a comma-separated collection of one or more *type-name list items*. A directive-name list consists of a comma-separated collection of one or more *directive-name list items*, each of which is the *directive-name* of some OpenMP directive. A foreign runtime preference list consists of a comma-separated collection of one or more *foreign-runtime list items* each of which is an OpenMP *foreign-runtime* identifier; the order of list items on a foreign runtime preference list is significant.

C / C++

A *variable list item* is a variable or an array section. An *extended list item* is a *variable list item* or a function name. A *locator list item* is any lvalue expression including variables, array sections, and reserved locators. A *parameter list item* is the name of a function parameter. A *type-name list item* is a type name.

C / C++

Fortran

A *variable list item* is one of the following:

- a variable that is not coindexed and that is not a substring;
- an array section that is not coindexed and that does not contain an element that is a substring;
- a *named constant*;
- an associate name that may appear in a variable definition context; or
- a common block name (enclosed in slashes).

An *extended list item* is a *variable list item* or a procedure name. A *locator list item* is a *variable list item*, or a reserved locator. A *parameter list item* is a dummy argument of a subroutine or function. A *type-name list item* is a type specifier that must not be **CLASS(*)** or an abstract type.

A *named constant* as a *list item* can appear only in clauses where it is explicitly allowed.

When a named common block appears in an OpenMP argument list, it has the same meaning and restrictions as if every explicit member of the common block appeared in the list. An explicit member of a common block is a variable that is named in a **COMMON** statement that specifies the common block name and is declared in the same scoping unit in which the clause appears. Named common blocks do not include the blank common block.

Although variables in common blocks can be accessed by use association or host association, common block names cannot. As a result, a common block name specified in a data-sharing attribute, a data copying, or a data-mapping attribute clause must be declared to be a common block in the same scoping unit in which the clause appears.

If a list item that appears in a directive or clause is an optional dummy argument that is not present, the directive or clause for that list item is ignored.

If the variable referenced inside a construct is an optional dummy argument that is not present, any explicitly determined, implicitly determined, or predetermined data-sharing and data-mapping attribute rules for that variable are ignored. Otherwise, if the variable is an optional dummy argument that is present, it is present inside the construct.

Fortran

Restrictions

The restrictions to OpenMP lists are as follows:

- Unless otherwise specified, OpenMP list items must be directive-wide unique, i.e., a list item can only appear once in one OpenMP list of all arguments, clauses, and modifiers of the directive.

C

- Unless otherwise specified, a variable that is part of another variable (as an array element or a structure element) cannot be a variable list item, an extended list item or a locator list item.

C

C++

- Unless otherwise specified, a variable that is part of another variable (as an array element or a structure element) cannot be a variable list item, an extended list item or locator list item except if the list appears on a clause that is associated with a construct within a class non-static member function and the variable is an accessible data member of the object for which the non-static member function is invoked.

C++

Fortran

- Unless otherwise specified, a variable that is part of another variable (as an array or structure element) cannot be a variable list item, an extended list item or locator list item.

Fortran

3.2.2 Reserved Locators

On some directives, some clauses accept the use of reserved locators as special identifiers that represent system storage not necessarily bound to any base language storage item. Reserved locators may only appear in clauses and directives where they are explicitly allowed and may not otherwise be referenced in the program. The list of reserved locators is:

```
omp_all_memory
```

The reserved locator **omp_all_memory** is a reserved identifier that denotes a list item treated as having storage that corresponds to the storage of all other objects in memory.

C / C++

3.2.3 Array Shaping

If an expression has a type of pointer to *T*, then a shape-operator can be used to specify the extent of that pointer. In other words, the shape-operator is used to reinterpret, as an n-dimensional array, the region of memory to which that expression points.

Formally, the syntax of the shape-operator is as follows:

shaped-expression **:=** **([**s_1**][**s_2**]...[**s_n**])** *cast-expression*

The result of applying the shape-operator to an expression is an lvalue expression with an n-dimensional array type with dimensions $s_1 \times s_2 \ldots \times s_n$ and element type *T*.

The precedence of the shape-operator is the same as a type cast.

Each s_i is an integral type expression that must evaluate to a positive integer.

Restrictions

Restrictions to the shape-operator are as follows:

- The type *T* must be a complete type.
- The shape-operator can appear only in clauses for which it is explicitly allowed.
- The result of a shape-operator must be a named array of a list item.
- The type of the expression upon which a shape-operator is applied must be a pointer type.

C++

- If the type *T* is a reference to a type *T'*, then the type will be considered to be *T'* for all purposes of the designated array.

C++

C / C++

3.2.4 Array Sections

An array section designates a subset of the elements in an array.

C / C++

To specify an array section in an OpenMP directive, array subscript expressions are extended with the following syntax:

```
[ lower-bound : length : stride] or
[ lower-bound : length : ] or
[ lower-bound : length ] or
[ lower-bound : : stride] or
[ lower-bound : : ] or
[ lower-bound : ] or
[  : length : stride] or
[  : length : ] or
[  : length ] or
[ : : stride]
[ : : ]
[ : ]
```

The array section must be a subset of the original array.

Array sections are allowed on multidimensional arrays. Base language array subscript expressions can be used to specify length-one dimensions of multidimensional array sections.

Each of the *lower-bound*, *length*, and *stride* expressions if specified must be an integral type *expression* of the base language. When evaluated they represent a set of integer values as follows:

{ *lower-bound*, *lower-bound* + *stride*, *lower-bound* + 2 * *stride*,... , *lower-bound* + ((*length* - 1) * *stride*) }

The *length* must evaluate to a non-negative integer.

The *stride* must evaluate to a positive integer.

When the size of the array dimension is not known, the *length* must be specified explicitly.

When the *stride* is absent it defaults to 1.

When the *length* is absent it defaults to $\lceil (size - lower\text{-}bound)/stride \rceil$, where *size* is the size of the array dimension.

When the *lower-bound* is absent it defaults to 0.

The precedence of a subscript operator that uses the array section syntax is the same as the precedence of a subscript operator that does not use the array section syntax.

Note – The following are examples of array sections:

```
a[0:6]
a[0:6:1]
a[1:10]
a[1:]
a[:10:2]
b[10][:][:]
b[10][:][:0]
c[42][0:6][:]
c[42][0:6:2][:]
c[1:10][42][0:6]
S.c[:100]
p->y[:10]
this->a[:N]
(p+10)[:N]
```

Assume **a** is declared to be a 1-dimensional array with dimension size 11. The first two examples are equivalent, and the third and fourth examples are equivalent. The fifth example specifies a stride of 2 and therefore is not contiguous.

Assume **b** is declared to be a pointer to a 2-dimensional array with dimension sizes 10 and 10. The sixth example refers to all elements of the 2-dimensional array given by **b[10]**. The seventh example is a zero-length array section.

Assume **c** is declared to be a 3-dimensional array with dimension sizes 50, 50, and 50. The eighth example is contiguous, while the ninth and tenth examples are not contiguous.

The final four examples show array sections that are formed from more general base expressions.

The following are examples that are non-conforming array sections:

```
s[:10].x
p[:10]->y
*(xp[:10])
```

For all three examples, a base language operator is applied in an undefined manner to an array section. The only operator that may be applied to an array section is a subscript operator for which the array section appears as the postfix expression.

C / C++

Fortran

Fortran has built-in support for array sections although some restrictions apply to their use in OpenMP directives, as enumerated in the following section.

Fortran

Restrictions

Restrictions to array sections are as follows:

- An array section can appear only in clauses for which it is explicitly allowed.
- A *stride* expression may not be specified unless otherwise stated.

C / C++

- An element of an array section with a non-zero size must have a complete type.
- The base expression of an array section must have an array or pointer type.
- If a consecutive sequence of array subscript expressions appears in an array section, and the first subscript expression in the sequence uses the extended array section syntax defined in this section, then only the last subscript expression in the sequence may select array elements that have a pointer type.

C / C++

C++

- If the type of the base expression of an array section is a reference to a type *T*, then the type will be considered to be *T* for all purposes of the array section.
- An array section cannot be used in an overloaded `[]` operator.

C++

Fortran

- If a stride expression is specified, it must be positive.
- The upper bound for the last dimension of an assumed-size dummy array must be specified.
- If a list item is an array section with vector subscripts, the first array element must be the lowest in the array element order of the array section.
- If a list item is an array section, the last *part-ref* of the list item must have a section subscript list.

Fortran

3.2.5 Iterators

An iterator modifier is a unique, complex modifier that defines iterators and their values. An iterator is an identifier that expands to those multiple values in the argument for which it is specified. The *modifier-parameter-specification* of an iterator modifier is an *iterators-definition* with this format:

```
iterator-specifier [, iterators-definition ]
```

where *iterator-specifier* is:

C / C++

```
[ iterator-type ] identifier = range-specification
```

C / C++

Fortran

```
[ iterator-type :: ] identifier = range-specification
```

Fortran

where:

- *identifier* is a base language identifier.
- *iterator-type* is a type-name list item.
- *range-specification* is of the form *begin*:*end*[:*step*], where *begin* and *end* are expressions for which their types can be converted to *iterator-type* and *step* is an integral expression.

C / C++

In an *iterator-specifier*, if the *iterator-type* is not specified then that iterator is of **`int`** type.

C / C++

Fortran

In an *iterator-specifier*, if the *iterator-type* is not specified then that iterator has default integer type.

Fortran

In a *range-specification*, if the *step* is not specified its value is implicitly defined to be 1.

An iterator only exists in the context of the clause argument in which it appears. An iterator also hides all accessible symbols with the same name in the context of the clause argument.

The use of a variable in an expression that appears in the *range-specification* causes an implicit reference to the variable in all enclosing constructs.

C / C++

The values of the iterator are the set of values $i_0, \ldots, i_{N-1}$ where:

- $i_0 = (\textit{iterator-type})\ begin$;
- $i_j = (\textit{iterator-type})\ (i_{j-1} + step)$, where $j \geq 1$; and
- if $step > 0$,
  - $i_0 < (\textit{iterator-type})\ end$;
  - $i_{N-1} < (\textit{iterator-type})\ end$; and
  - $(\textit{iterator-type})\ (i_{N-1} + step) \geq (\textit{iterator-type})\ end$;
- if $step < 0$,
  - $i_0 > (\textit{iterator-type})\ end$;
  - $i_{N-1} > (\textit{iterator-type})\ end$; and
  - $(\textit{iterator-type})\ (i_{N-1} + step) \leq (\textit{iterator-type})\ end$.

C / C++

Fortran

The values of the iterator are the set of values $i_1, \ldots, i_N$ where:

- $i_1 = begin$;
- $i_j = i_{j-1} + step$, where $j \geq 2$; and
- if $step > 0$,
  - $i_1 \leq end$;
  - $i_N \leq end$; and
  - $i_N + step > end$;
- if $step < 0$,
  - $i_1 \geq end$;
  - $i_N \geq end$; and
  - $i_N + step < end$.

Fortran

The set of values will be empty if no possible value complies with the conditions above.

For those arguments that contain expressions that contain iterator identifiers, the effect is as if the list item is instantiated within the clause for each value of the iterator in the set defined above, substituting each occurrence of the iterator identifier in the expression with the iterator value. If the set of values of the iterator is empty then the effect is as if the clause was not specified.

The behavior is unspecified if $i_j + step$ cannot be represented in *iterator-type* in any of the $i_j + step$ computations for any $0 \leq j < N$ in C/C++ or $0 < j \leq N$ in Fortran.

Restrictions

Restrictions to iterators are as follows:

- An expression that contains an iterator identifier can only appear in clauses that explicitly allow expressions that contain iterators.
- The *iterator-type* must not declare a new type.

C / C++

- The *iterator-type* must be an integral or pointer type.
- The *iterator-type* must not be **`const`** qualified.

C / C++

Fortran

- The *iterator-type* must be an integer type.

Fortran

- If the *step* expression of a *range-specification* equals zero, the behavior is unspecified.
- Each iterator identifier can only be defined once in an *iterators-definition*.
- Iterators cannot appear in the *range-specification*.

3.3 Conditional Compilation

In implementations that support a preprocessor, the **`_OPENMP`** macro name is defined to have the decimal value *yyyymm* where *yyyy* and *mm* are the year and month designations of the version of the OpenMP API that the implementation supports.

If a **`#define`** or a **`#undef`** preprocessing directive in user code defines or undefines the **`_OPENMP`** macro name, the behavior is unspecified.

Fortran

The OpenMP API requires Fortran lines to be compiled conditionally, as described in the following sections.

Fortran

Fortran

3.3.1 Fixed Source Form Conditional Compilation Sentinels

The following conditional compilation sentinels are recognized in fixed form source files:

```
xtnormal{|} *$ | c$
```

To enable conditional compilation, a line with a conditional compilation sentinel must satisfy the following criteria:

- The sentinel must start in column 1 and appear as a single word with no intervening white space;
- After the sentinel is replaced with two spaces, initial lines must have a space or zero in column 6 and only white space and numbers in columns 1 through 5;
- After the sentinel is replaced with two spaces, continuation lines must have a character other than a space or zero in column 6 and only white space in columns 1 through 5.

If these criteria are met, the sentinel is replaced by two spaces. If these criteria are not met, the line is left unchanged.

Note – In the following example, the two forms for specifying conditional compilation in fixed source form are equivalent (the first line represents the position of the first 9 columns):

```
c23456789
!$ 10 iam = omp_get_thread_num() +
!$   &          index

#ifdef _OPENMP
   10 iam = omp_get_thread_num() +
     &          index
#endif
```

Fortran

Fortran

3.3.2 Free Source Form Conditional Compilation Sentinel

The following conditional compilation sentinel is recognized in free form source files:

To enable conditional compilation, a line with a conditional compilation sentinel must satisfy the following criteria:

- The sentinel can appear in any column but must be preceded only by white space;

- The sentinel must appear as a single word with no intervening white space;
- Initial lines must have a space after the sentinel;
- Continued lines must have an ampersand as the last non-blank character on the line, prior to any comment appearing on the conditionally compiled line.

Continuation lines can have an ampersand after the sentinel, with optional white space before and after the ampersand. If these criteria are met, the sentinel is replaced by two spaces. If these criteria are not met, the line is left unchanged.

Note – In the following example, the two forms for specifying conditional compilation in free source form are equivalent (the first line represents the position of the first 9 columns):

```
c23456789
 !$ iam = omp_get_thread_num() +      &
 !$&    index

#ifdef _OPENMP
    iam = omp_get_thread_num() +      &
        index
#endif
```

Fortran

3.4 if Clause

Name:	**Properties:**
if	default

Arguments:

Name	Type	Properties
if-expression	Expression of type logical	default

Modifiers:

Name	Modifies	Type	Properties
directive-name-modifier	*if-expression*	**Keyword: directive-name**	unique

Directives:

cancel, **parallel**, **simd**, **target**, **target data**, **target enter data**, **target exit data**, **target update**, **task**, **taskloop**

Semantics

If no *directive-name-modifier* is specified then the effect is as if a *directive-name-modifier* was specified with the *directive-name* of the directive on which the clause appears.

The effect of the **if** clause depends on the construct to which it is applied. If the construct is not a combined or composite construct then the effect is described in the section that describes that construct. For combined or composite constructs, the **if** clause only applies to the semantics of the construct named in the *directive-name-modifier*. For a combined or composite construct. If the *directive-name* of that construct is specified for the *directive-name-modifier* then the **if** clause applies to all constructs to which an **if** clause can apply.

Restrictions

Restrictions to the **if** clause are as follows:

- At most one **if** clause can be specified that applies to the semantics of any construct or constituent construct of a *directive-specification*.
- The *directive-name-modifier* must specify the *directive-name* of the construct or of a constituent construct of the *directive-specification* on which the **if** clause appears.

3.5 `destroy` Clause

Name:	**Properties:**
destroy	unique

Arguments:

Name	Type	Properties
destroy-var	Variable of type OpenMP variable	default

Directives:

depobj, **interop**

Additional information: When the **destroy** clause appears on the **depobj** construct, the *destroy-var* argument may be omitted. This syntax has been deprecated.

Semantics

If the **destroy** clause appears on a **debobj** construct and *destroy-var* is not specified, the effect is as if *destroy-var* refers to the same OpenMP depend object as the *depobj* argument of the construct. The syntax of the **destroy** clause on the **depbj** construct that does not specify *destroy-var* has been deprecated. When the **destroy** clause appears on a **depobj** construct, the state of *destroy-var* is set to uninitialized.

When the **destroy** clause appears on an **interop** construct, the *interop-type* is inferred based on the *interop-type* used to initialize *destroy-var* and *destroy-var* is set to the value of **omp_interop_none** after resources associated with *destroy-var* are released. The object referred to by *destroy-var* is unusable after destruction and the effect of using values associated with it is unspecified until it is initialized again by another **interop** construct.

Restrictions

- *destroy-var* must be non-const.
- If the **destroy** clause appears on a **depobj** construct *destroy-var* must refer to the same depend object as the *depobj* argument of the construct.
- If the **destroy** clause appears on an **interop** construct *destroy-var* must refer to a variable of OpenMP **interop** type.

Cross References

- **interop** construct, see Section 14.1.
- **depobj** construct, see Section 15.9.4.

4 Base Language Formats and Restrictions

This section defines concepts and restrictions on base language code used in OpenMP. The concepts help support base language neutrality for OpenMP directives and their associated semantics.

Restrictions

The following restrictions apply generally for base language code in an OpenMP program:

- Programs must not declare names that begin with the **omp_** or **ompx_** prefix, as these are reserved for the OpenMP implementation.

C++

- Programs must not declare a namespace with the **omp** or **ompx** names, as these are reserved for the OpenMP implementation.

C++

4.1 OpenMP Types and Identifiers

An OpenMP identifier is a special identifier for use within OpenMP directives and clauses for some specific purpose. For example, OpenMP reduction identifiers specify the combiner operation to use in a reduction, OpenMP mapper identifiers specify the name of a user-defined mapper, and OpenMP foreign runtime identifiers specify the name of a foreign runtime.

Generic OpenMP types specify the type of expression or variable that is used in OpenMP contexts regardless of the base language. These types support the definition of many important OpenMP concepts independently of the base language in which they are used.

The assignable OpenMP type instance is defined to facilitate base language neutrality. An assignable OpenMP type instance can be used as an argument of an OpenMP construct in order for the implementation to modify the value of that instance.

C / C++

An assignable OpenMP type instance is an lvalue expression of that OpenMP type.

C / C++

Fortran

An assignable OpenMP type instance is a variable of that OpenMP type.

Fortran

The OpenMP logical type supports logical variables and expressions in any base language.

C / C++

Any OpenMP logical expression is a scalar expression. This document uses *true* as a generic term for a non-zero integer value and *false* as a generic term for an integer value of zero.

C / C++

Fortran

Any OpenMP logical expression is a scalar logical expression. This document uses *true* as a generic term for a logical value of **.TRUE.** and *false* as a generic term for a logical value of **.FALSE.**.

Fortran

The OpenMP integer type supports integer variables and expressions in any base language.

C / C++

Any OpenMP integer expression is an integer expression.

C / C++

Fortran

Any OpenMP integer expression is a scalar integer expression.

Fortran

The OpenMP string type supports character string variables and expressions in any base language.

C / C++

Any OpenMP string expression is an expression of **char *** type.

C / C++

Fortran

Any OpenMP string expression is a character string of default kind.

Fortran

OpenMP function identifiers support function names in any base language. Regardless of the base language, any OpenMP function identifier is the name of a function as a base language identifier.

Each OpenMP type other than those specifically defined in this section has a generic name, *<generic_name>*, by which it is referred throughout this document and that is used to construct the base language construct that corresponds to that OpenMP type.

C / C++

A variable of OpenMP type with name *<generic_name>* is a variable of type **omp_***<generic_name>***_t**.

C / C++

Fortran

A variable of OpenMP type with name *<generic_name>* is a scalar integer variable of kind **omp_***<generic_name>***_kind**.

Fortran

Cross References

- OpenMP mapper idenitifiers, see Section 5.8.1.
- OpenMP foreign runtime idenitifiers, see Section 14.1.1.
- OpenMP reduction idenitifiers, see Section 5.5.1.

4.2 OpenMP Stylized Expressions

An OpenMP stylized expression is a base language expressions that is subject to restrictions that enable its use within an OpenMP implementation. These expressions often make use of special variable identifiers that the implementation binds to well-defined internal state.

Cross References

- OpenMP combiner expressions, see Section 5.5.2.
- OpenMP initializer expressions, see Section 5.5.3.

4.3 Structured Blocks

This section specifies the concept of a structured block. A structured block:

- may contain infinite loops where the point of exit is never reached;
- may halt due to an IEEE exception;

C / C++

- may contain calls to **exit()**, **_Exit()**, **quick_exit()**, **abort()** or functions with a **_Noreturn** specifier (in C) or a **noreturn** attribute (in C/C++);
- may be an expression statement, iteration statement, selection statement, or try block, provided that the corresponding compound statement obtained by enclosing it in **{** and **}** would be a structured block; and

C / C++

Fortran

- may contain **STOP** or **ERROR STOP** statements.

Fortran

C / C++

A structured block sequence that consists of more than one statement may appear only for executable directives that explicitly allow it. The corresponding compound statement obtained by enclosing the sequence in **{** and **}** must be a structured block and the structured block sequence then should be considered to be a structured block with all of its restrictions.

C / C++

Restrictions

Restrictions to structured blocks are as follows:

- Entry to a structured block must not be the result of a branch.
- The point of exit cannot be a branch out of the structured block.

C / C++

- The point of entry to a structured block must not be a call to **setjmp**.
- **longjmp** must not violate the entry/exit criteria.

C / C++

C++

- **throw** must not violate the entry/exit criteria of structured blocks.
- **co_await**, **co_yield** and **co_return** must not violate the entry/exit criteria of structured blocks.

C++

Fortran

- When a **BLOCK** construct appears in a structured block, that **BLOCK** construct must not contain any **ASYNCHRONOUS** or **VOLATILE** statements, nor any specification statements that include the **ASYNCHRONOUS** or **VOLATILE** attributes.

Fortran

4.3.1 OpenMP Context-Specific Structured Blocks

An OpenMP context-specific structured block consists of statements that conform to specific restrictions so that OpenMP can treat them as a structured block or a structured block sequence. The restrictions depend on the context in which the context-specific structured block can be used.

Fortran

An OpenMP *allocator structured block* consists of *allocate-stmt*, where *allocate-stmt* is a Fortran **ALLOCATE** statement. Allocator structured blocks are considered strictly structured blocks for the purpose of the **allocators** construct.

Fortran

Cross References

- **allocators** construct, see Section 6.8.

4.3.2 OpenMP Function Dispatch Structured Blocks

An OpenMP *function dispatch structured block* is a context-specific structured block that identifies the location of a function dispatch.

C / C++

A function dispatch structured block is is an expression statement the following form:

```
expression = target-call ( [expression-list] );
target-call ( [expression-list] );
```

C / C++

Fortran

A function dispatch structured block is is an expression statement the following form:

```
expression = target-call ( [arguments] )
CALL target-call [ ( [arguments] ) ]
```

For purposes of the **dispatch** construct, the expression statement is considered a strictly structured block.

Fortran

Restrictions

Restrictions to the function dispatch structured blocks are as follows:

C++

- The *target-call* expression can only be a direct call.

C++

Fortran

- *target-call* must be a procedure name.
- *target-call* must not be a procedure pointer.

Fortran

Cross References

- **dispatch** construct, see Section 7.6.

4.3.3 OpenMP Atomic Structured Blocks

An OpenMP *atomic structured block* is a context-specific structured block that can appear in an **`atomic`** construct. The form of an atomic structured block depends on the atomic semantics that the directive enforces.

In the following definitions:

C / C++

- *x*, *r* (result), and *v* (as applicable) are *lvalue* expressions with scalar type.
- *e* (expected) is an expression with scalar type,
- *d* (desired) is an expression with scalar type.
- *e* and *v* may refer to, or access, the same storage location.
- *expr* is an expression with scalar type.
- The order operation, *ordop*, is one of `<`, or `>`.
- *binop* is one of `+`, `*`, `-`, `/`, `&`, `^`, `|`, `<<`, or `>>`.
- `==` comparisons are performed by comparing the bits that comprise each object as with **`memcmp`**.
- For forms that allow multiple occurrences of *x*, the number of times that *x* is evaluated is unspecified.

C / C++

Fortran

- *x*, *v*, *d* and *e* (as applicable) are scalar variables of intrinsic type.
- *expr* is a scalar expression.
- *expr-list* is a comma-separated, non-empty list of scalar expressions.
- *intrinsic-procedure-name* is one of **`MAX`**, **`MIN`**, **`IAND`**, **`IOR`**, or **`IEOR`**.
- *operator* is one of `+`, `*`, `-`, `/`, `.AND.`, `.OR.`, `.EQV.`, or `.NEQV.`.
- For forms that allow multiple occurrences of *x*, the number of times that *x* is evaluated is unspecified.

Fortran

A *read-atomic* structured block can be specified for **`atomic`** directives that enforce atomic read semantics but not capture semantics.

C / C++

A *read-atomic* structured block is *read-expr-stmt*, a read expression statement that has the following form:

```
v = x;
```

C / C++

Fortran

A *read-atomic* structured block is *read-statement*, a read statement that has the following form:

```
v = x;
```

Fortran

A *write-atomic* structured block can be specified for **atomic** directives that enforce atomic write semantics but not capture semantics.

C / C++

A *write-atomic* structured block is *write-expr-stmt*, a write expression statement that has the following form:

```
x = expr;
```

C / C++

Fortran

A *write-atomic* structured block is *write-statement*, a write statement that has the following form:

```
x = expr
```

Fortran

An *update-atomic* structured block can be specified for **atomic** directives that enforce atomic update semantics but not capture semantics.

C / C++

An *update-atomic* structured block is *update-expr-stmt*, an update expression statement that has one of the following forms:

```
x++;
x--;
++x;
--x;
x binop= expr;
x = x binop expr;
x = expr binop x;
```

C / C++

Fortran

An *update-atomic* structured block is *update-statement*, an update statement that has one of the following forms:

```
x = x operator expr
x = expr operator x
x = intrinsic-procedure-name (x, expr-list)
x = intrinsic-procedure-name (expr-list, x)
```

Fortran

A *conditional-update-atomic* structured block can be specified for **atomic** directives that enforce atomic conditional update semantics but not capture semantics.

C / C++

A *conditional-update-atomic* structured block is either *cond-expr-stmt*, a conditional expression statement that has one of the following forms:

```
x = expr ordop x ? expr : x;
x = x ordop expr ? expr : x;
x = x == e ? d : x;
```

or *cond-update-stmt*, a conditional update statement that has one of the following forms:

```
if(expr ordop x) { x = expr; }
if(x ordop expr) { x = expr; }
if(x == e) { x = d; }
```

C / C++

Fortran

A *conditional-update-atomic* structured block is *conditional-update-statement*, a conditional update statement that has one of the following forms:

```
if (x == e) then
  x = d
end if
```

or

```
if (x == e) x = d
```

read-atomic, *write-atomic*, *update-atomic*, and *conditional-update-atomic* structured blocks are considered strictly structured blocks for the purpose of the **atomic** construct.

Fortran

A *capture-atomic* structured block can be specified for **atomic** directives that enforce capture semantics. They are further catergorized as *write-capture-atomic*, *update-capture-atomic*, and *conditional-update-capture-atomic* structured blocks, which can be specified for **atomic** directives that enforce write, update or conditional update atomic semantics in addition to capture semantics.

C / C++

A *capture-atomic* structured block is *capture-stmt*, a capture statement that has one of the following forms:

```
v = expr-stmt
{ v = x; expr-stmt }
{ expr-stmt v = x; }
```

If *expr-stmt* is *write-expr-stmt* or *expr-stmt* is *update-expr-stmt* as specified above then it is an *update-capture-atomic* structured block. If *expr-stmt* is *cond-expr-stmt* as specified above then it is a *conditional-update-capture-atomic* structured block. In addition, a *conditional-update-capture-atomic* structured block can have one of the following forms:

```
{ v = x; cond-update-stmt }
{ cond-update-stmt v = x;  }
if(x == e) { x = d; } else { v = x; }
{ r = x == e; if(r) { x = d; } }
{ r = x == e; if(r) { x = d; } else { v = x; } }
```

C / C++

Fortran

A *capture-atomic* structured block has one of the following forms:

```
statement
capture-statement
```

or

```
capture-statement
statement
```

where *capture-statement* has the following form:

```
v = x
```

If *statement* is *write-statement* as specified above then it is a *write-capture-atomic* structured block. If *statement* is *update-statement* as specified above then it is an *update-capture-atomic* structured block. If *statement* is *conditonal-update-statement* as specified above then it is a *conditional-update-capture-atomic* structured block. In addition, for a *conditional-update-capture-atomic* structured block, *statement* can have the following form:

```
x = expr
```

In addition, a *conditional-update-capture-atomic* structured block can have the following form:

```
if (x == e) then
  x = d
else
  v = x
end if
```

All *capture-atomic* structured blocks are considered loosely structured blocks for the purpose of the **atomic** construct.

Fortran

Restrictions

Restrictions to OpenMP atomic structured blocks are as follows:

C / C++

- In forms where *e* is assigned it must be an lvalue.
- *r* must be of integral type.
- During the execution of an **atomic** region, multiple syntactic occurrences of *x* must designate the same storage location.
- None of *v*, *x*, *r*, *d* and *expr* (as applicable) may access the storage location designated by any other in the list.
- *binop*, *binop*=, *ordop*, **==**, **++**, and **--** are not overloaded operators.
- The expression *x binop expr* must be numerically equivalent to *x binop (expr)*. This requirement is satisfied if the operators in *expr* have precedence greater than *binop*, or by using parentheses around *expr* or subexpressions of *expr*.
- The expression *expr binop x* must be numerically equivalent to *(expr) binop x*. This requirement is satisfied if the operators in *expr* have precedence equal to or greater than *binop*, or by using parentheses around *expr* or subexpressions of *expr*.

C / C++

Fortran

- *x* must not have the **ALLOCATABLE** attribute.
- During the execution of an atomic region, multiple syntactic occurrences of *x* must designate the same storage location.
- None of *v*, *expr*, and *expr-list* (as applicable) may access the same storage location as *x*.
- None of *x*, *expr*, and *expr-list* (as applicable) may access the same storage location as *v*.
- If *intrinsic-procedure-name* refers to **IAND**, **IOR**, or **IEOR**, exactly one expression must appear in *expr-list*.
- The expression *x operator expr* must be numerically equivalent to *x operator (expr)*. This requirement is satisfied if the operators in *expr* have precedence greater than *operator*, or by using parentheses around *expr* or subexpressions of *expr*.
- The expression *expr operator x* must be numerically equivalent to *(expr) operator x*. This requirement is satisfied if the operators in *expr* have precedence equal to or greater than *operator*, or by using parentheses around *expr* or subexpressions of *expr*.

- *intrinsic-procedure-name* must refer to the intrinsic procedure name and not to other program entities.
- *operator* must refer to the intrinsic operator and not to a user-defined operator.
- All assignments must be intrinsic assignments.

Fortran

Cross References

- **atomic** construct, see Section 15.8.4.

4.4 Loop Concepts

OpenMP semantics frequently involve loops that occur in the base language code. As detailed in this section, OpenMP defines several concepts that facilitate the specification of those semantics and their associated syntax.

4.4.1 Canonical Loop Nest Form

A loop nest has *canonical loop nest form* if it conforms to *loop-nest* in the following grammar:

Symbol	Meaning
loop-nest	One of the following:

C / C++

```
for (init-expr; test-expr; incr-expr)
    loop-body
```

C / C++

or

C++

```
for (range-decl: range-expr)
    loop-body
```

A range-based **for** loop is equivalent to a regular **for** loop using iterators, as defined in the base language. A range-based **for** loop has no iteration variable.

C++

or

Fortran

```
DO [ label ] var = lb , ub [ , incr ]
   [intervening-code]
   loop-body
   [intervening-code]
[ label ] END DO
```

If the *loop-nest* is a *nonblock-do-construct*, it is treated as a *block-do-construct* for each **DO** construct.

The value of *incr* is the increment of the loop. If not specified, its value is assumed to be 1.

Fortran

or

loop-transformation-construct

or

generated-canonical-loop

loop-body One of the following:

loop-nest

or

C / C++

```
{
   [intervening-code]
   loop-body
   [intervening-code]
}
```

C / C++

or

Fortran

```
BLOCK
   [intervening-code]
   loop-body
   [intervening-code]
END BLOCK
```

Fortran

or if none of the previous productions match

final-loop-body

loop-transformation-construct A loop transformation construct.

generated-canonical-loop A generated loop from a loop transformation construct that has canonical loop nest form and for which the loop body matches *loop-body*.

intervening-code A structured block sequence that does not contain OpenMP directives or calls to the OpenMP runtime API in its corresponding region, referred to as intervening code. If intervening code is present, then a loop at the same depth within the loop nest is not a perfectly nested loop.

C / C++

It must not contain iteration statements, **continue** statements or **break** statements that apply to the enclosing loop.

C / C++

Fortran

It must not contain loops, array expressions, **CYCLE** statements or **EXIT** statements.

Fortran

final-loop-body A structured block that terminates the scope of loops in the loop nest. If the loop nest is associated with a loop-associated directive, loops in this structured block cannot be associated with that directive.

C / C++

init-expr One of the following:

var = lb

integer-type var = lb

C

pointer-type var = lb

C

C++

random-access-iterator-type var = lb

C++

test-expr One of the following:

var relational-op ub

ub relational-op var

relational-op One of the following:

<

<=

>

>=

!=

incr-expr One of the following:

++*var*

var++

- - *var*

var - -

var += *incr*

var - = *incr*

var = *var* + *incr*

var = *incr* + *var*

var = *var* - *incr*

The value of *incr*, respectively 1 and -1 for the increment and decrement operators, is the increment of the loop.

C / C++

var One of the following:

C / C++

A variable of a signed or unsigned integer type.

C / C++

C

A variable of a pointer type.

C

C++

A variable of a random access iterator type.

C++

Fortran

A variable of integer type.

Fortran

var is the iteration variable of the loop. It must not be modified during the execution of *intervening-code* or *loop-body* in the loop.

lb, *ub* One of the following:

Expressions of a type compatible with the type of *var* that are loop invariant with respect to the outermost loop.

or

One of the following:

var-outer
var-outer + *a2*
a2 + *var-outer*
var-outer - *a2*

where *var-outer* is of a type compatible with the type of *var*.

or

If *var* is of an integer type, one of the following:

a2 - *var-outer*
a1 * *var-outer*
a1 * *var-outer* + *a2*
a2 + *a1* * *var-outer*
a1 * *var-outer* - *a2*
a2 - *a1* * *var-outer*
var-outer * *a1*
var-outer * *a1* + *a2*
a2 + *var-outer* * *a1*
var-outer * *a1* - *a2*
a2 - *var-outer* * *a1*

where *var-outer* is of an integer type.

lb and *ub* are loop bounds. A loop for which *lb* or *ub* refers to *var-outer* is a non-rectangular loop. If *var* is of an integer type, *var-outer* must be of an integer type with the same signedness and bit precision as the type of *var*.

The coefficient in a loop bound is 0 if the bound does not refer to *var-outer*. If a loop bound matches a form in which *a1* appears, the coefficient is -*a1* if the product of *var-outer* and *a1* is subtracted from *a2*, and otherwise the coefficient is *a1*. For other matched forms where *a1* does not appear, the coefficient is -1 if *var-outer* is subtracted from *a2*, and otherwise the coefficient is 1.

a1, *a2*, *incr* Integer expressions that are loop invariant with respect to the outermost loop of the loop nest.

If the loop is associated with a loop-associated directive, the expressions are evaluated before the construct formed from that directive.

var-outer The loop iteration variable of a surrounding loop in the loop nest.

C++

range-decl A declaration of a variable as defined by the base language for range-based **`for`** loops.

range-expr An expression that is valid as defined by the base language for range-based **`for`** loops. It must be invariant with respect to the outermost loop of the loop nest and the iterator derived from it must be a random access iterator.

C++

Restrictions

Restrictions to canonical loop nests are as follows:

C / C++

- If *test-expr* is of the form *var relational-op b* and *relational-op* is < or <= then *incr-expr* must cause *var* to increase on each iteration of the loop. If *test-expr* is of the form *var relational-op b* and *relational-op* is > or >= then *incr-expr* must cause *var* to decrease on each iteration of the loop. Increase and decrease are using the order induced by *relational-op*.
- If *test-expr* is of the form *ub relational-op var* and *relational-op* is < or <= then *incr-expr* must cause *var* to decrease on each iteration of the loop. If *test-expr* is of the form *ub relational-op var* and *relational-op* is > or >= then *incr-expr* must cause *var* to increase on each iteration of the loop. Increase and decrease are using the order induced by *relational-op*.
- If *relational-op* is **`!=`** then *incr-expr* must cause *var* to always increase by 1 or always decrease by 1 and the increment must be a constant expression.
- *final-loop-body* must not contain any **`break`** statement that would cause the termination of the innermost loop.

C / C++

Fortran

- *final-loop-body* must not contain any **`EXIT`** statement that would cause the termination of the innermost loop.

Fortran

- A *loop-nest* must also be a structured block.
- For a non-rectangular loop, if *var-outer* is referenced in *lb* and *ub* then they must both refer to the same iteration variable.
- For a non-rectangular loop, let a_{lb} and a_{ub} be the respective coefficients in *lb* and *ub*, $incr_{\mathrm{inner}}$ the increment of the non-rectangular loop and $incr_{\mathrm{outer}}$ the increment of the loop referenced by *var-outer*. $incr_{\mathrm{inner}}(a_{\mathrm{ub}} - a_{\mathrm{lb}})$ must be a multiple of $incr_{\mathrm{outer}}$.
- The loop iteration variable may not appear in a **`threadprivate`** directive.

Cross References

- Loop transformation constructs, see Section 9.
- **`threadprivate`** directive, see Section 5.2.

4.4.2 OpenMP Loop-Iteration Spaces and Vectors

A loop-associated directive controls some number of the outermost loops of an associated loop nest, called the associated loops, in accordance with its specified clauses. These associated loops and their loop iteration variables form an OpenMP *loop-iteration space*. OpenMP *loop-iteration vectors* allow other directives to refer to points in that loop-iteration space.

A loop transformation construct that appears inside a loop nest is replaced according to its semantics before any loop can be associated with a loop-associated directive that is applied to the loop nest. The depth of the loop nest is determined according to the loops in the loop nest, after any such replacements have taken place. A loop counts towards the depth of the loop nest if it is a base language loop statement or generated loop and it matches *loop-nest* while applying the production rules for canonical loop nest form to the loop nest.

The canonical loop nest form allows the iteration count of all associated loops to be computed before executing the outermost loop.

For any associated loop, the iteration count is computed as follows:

C / C++

- If *var* has a signed integer type and the *var* operand of *test-expr* after usual arithmetic conversions has an unsigned integer type then the loop iteration count is computed from *lb*, *test-expr* and *incr* using an unsigned integer type corresponding to the type of *var*.
- Otherwise, if *var* has an integer type then the loop iteration count is computed from *lb*, *test-expr* and *incr* using the type of *var*.

C / C++

C

- If *var* has a pointer type then the loop iteration count is computed from *lb*, *test-expr* and *incr* using the type **`ptrdiff_t`**.

C

C++

- If *var* has a random access iterator type then the loop iteration count is computed from *lb*, *test-expr* and *incr* using the type **`std::iterator_traits`**<*random-access-iterator-type*>**`::difference_type`**.
- For range-based **`for`** loops, the loop iteration count is computed from *range-expr* using the type **`std::iterator_traits`**<*random-access-iterator-type*>**`::difference_type`** where *random-access-iterator-type* is the iterator type derived from *range-expr*.

C++

Fortran

- The loop iteration count is computed from *lb*, *ub* and *incr* using the type of *var*.

Fortran

The behavior is unspecified if any intermediate result required to compute the iteration count cannot be represented in the type determined above.

No synchronization is implied during the evaluation of the *lb*, *ub*, *incr* or *range-expr* expressions. Whether, in what order, or how many times any side effects within the *lb*, *ub*, *incr*, or *range-expr* expressions occur is unspecified.

Let the number of loops associated with a construct be n. The OpenMP loop-iteration space is the n-dimensional space defined by the values of var_i, $1 \le i \le n$, the iteration variables of the associated loops, with $i = 1$ referring to the outermost loop of the loop nest. An OpenMP loop-iteration vector, which may be used as an argument of OpenMP directives and clauses, then has the form:

$$var_1\ [\pm\ offset_1],\ var_2\ [\pm\ offset_2],\ \ldots,\ var_n\ [\pm\ offset_n]$$

where $offset_i$ is a compile-time constant non-negative OpenMP integer expression that facilitates identification of relative points in the loop-iteration space.

The iterations of some number of associated loops can be collapsed into one larger iteration space that is called the logical iteration space. The particular integer type used to compute the iteration count for the collapsed loop is implementation defined, but its bit precision must be at least that of the widest type that the implementation would use for the iteration count of each loop if it was the only associated loop. OpenMP defines a special loop-iteration vector, **`omp_cur_iteration`**, for which $offset_i = 0\ \forall\ i$. This loop-iteration vector enables identification of relative points in the logical iteration space as:

`omp_cur_iteration` *[± logical_offset]* where *logical_offset* is a compile-time constant non-negative OpenMP integer expression.

For directives that result in the execution of a collapsed logical iteration space, the number of times that any intervening code between any two loops of the same logical iteration space will be executed is unspecified but will be the same for all intervening code at the same depth, at least once per iteration of the loop that encloses the intervening code and at most once per logical iteration. If the iteration count of any loop is zero and that loop does not enclose the intervening code, the behavior is unspecified.

4.4.3 `collapse` Clause

Name: **`collapse`**

Properties: unique

Arguments:

Name	Type	Properties
n	Expression of type integer	default

Directives:

`distribute`, **`do`**, **`for`**, **`loop`**, **`simd`**, **`taskloop`**

Semantics

The **`collapse`** clause associates one or more loops with the directive on which it appears for the purpose of identifying the portion of the depth of the canonical loop nest to which to apply the semantics of the directive. The argument *n* specifies the number of loops of the associated loop nest to which to apply those semantics. On all directives on which the **`collapse`** clause may appear, the effect is as if a value of one was specified for *n* if the **`collapse`** clause is not specified.

Restrictions

- *n* must not evaluate to a value greater than the depth of the associated loop nest.

Cross References

- Worksharing-loop constructs, see Section 11.5.
- **`simd`** construct, see Section 10.4.
- **`do`** construct, see Section 11.5.2.
- **`for`** construct, see Section 11.5.1.
- **`ordered`** clause, see Section 4.4.4.

4.4.4 `ordered` Clause

Name:	Properties:
`ordered`	unique

Arguments:

Name	Type	Properties
n	Expression of type integer	optional, constant, positive

Directives:

`do`, `for`, `simd`

Semantics

The **`ordered`** clause associates one or more loops with the directive on which it appears for the purpose of identifying cross-iteration dependences. The argument *n* specifies the number of loops of the associated loop to use for that purpose. If *n* is not specified then the behavior is as if *n* is specified with the same value as is specified for the **`collapse`** clause on the construct.

Restrictions

- None of the associated loops may be non-rectangular loops.
- The **`ordered`** clause must not appear on a worksharing-loop directive if the associated loops include the generated loops of a **`tile`** directive.
- *n* must not evaluate to a value greater than the depth of the associated loop nest.
- If *n* is explicitly specified, the associated loops must be perfectly nested.
- If *n* is explicitly specified and the **`collapse`** clause is also specified for the **`ordered`** clause on the same construct, *n* must be greater than or equal to the *n* specified for the **`collapse`** clause.
- If *n* is explicitly specified, a **`linear`** clause must not be specified on the same directive.

C++

- If *n* is explicitly specified, none of the associated loops may be a range-based **`for`** loop.

C++

Cross References

- Worksharing-loop constructs, see Section 11.5.
- **`simd`** construct, see Section 10.4.
- **`collapse`** clause, see Section 4.4.3.
- **`do`** construct, see Section 11.5.2.
- **`for`** construct, see Section 11.5.1.
- **`linear`** clause, see Section 5.4.6.
- **`tile`** construct, see Section 9.1.

4.4.5 Consistent Loop Schedules

For constructs formed from loop-associated directives that have consistent schedules, the implementation will guarantee that memory effects of a logical iteration in the first loop nest happen before the execution of the same logical iteration in the second loop nest.

Two constructs formed from loop-associated directives have consistent schedules if all of the following conditions hold:

- The constructs are formed from directives with the same directive name;
- The regions that correspond to the two constructs have the same binding region;
- The constructs have the same reproducible schedule;
- The associated loop nests have identical logical iteration vector spaces; and
- The associated loop nests are either both rectangular or both non-rectangular.

5 Data Environment

This chapter presents directives and clauses for controlling data environments. These clauses and directives include the *data-environment attribute clauses*, which explicitly determine the attributes of variables identified in a *list* parameter. The data-environment attribute clauses form a general clause set for which certain restrictions apply to their use on directives that accept any members of the set. In addition, these clauses are divided into two subsets that also form general clause sets. Additional restrictions apply to the use of these sets, which are the *data-sharing attribute clauses* and the *data-mapping attribute clauses*, on directives that accept any members of them.

5.1 Data-Sharing Attribute Rules

This section describes how the data-sharing attributes of variables referenced in data environments are determined. The following two cases are described separately:

- Section 5.1.1 describes the data-sharing attribute rules for variables referenced in a construct.
- Section 5.1.2 describes the data-sharing attribute rules for variables referenced in a region, but outside any construct.

5.1.1 Variables Referenced in a Construct

The data-sharing attributes of variables that are referenced in a construct can be *predetermined*, *explicitly determined*, or *implicitly determined*, according to the rules outlined in this section.

Specifying a variable in a data-sharing attribute clause, except for the **`private`** clause, or **`copyprivate`** clause of an enclosed construct, causes an implicit reference to the variable in the enclosing construct. Specifying a variable in a **`map`** clause of an enclosed construct may cause an implicit reference to the variable in the enclosing construct. Such implicit references are also subject to the data-sharing attribute rules outlined in this section.

Fortran

A type parameter inquiry or complex part designator that is referenced in a construct is treated as if its designator is referenced.

Fortran

Certain variables and objects have *predetermined* data-sharing attributes for the construct in which they are referenced. The first matching rule from the following list of predetermined data-sharing attribute rules applies for variables and objects that are referenced in a construct.

C / C++

- Variables that appear in **threadprivate** directives or variables with the **_Thread_local** (in C) or **thread_local** (in C++) storage-class specifier are threadprivate.

C

- Variables with automatic storage duration that are declared in a scope inside the construct are private.

C

C++

- Variables of non-reference type with automatic storage duration that are declared in a scope inside the construct are private.

C++

- Objects with dynamic storage duration are shared.
- The loop iteration variable in any associated loop of a **for**, **parallel for**, **taskloop**, or **distribute** construct is private.
- The loop iteration variable in the associated loop of a **simd** construct with just one associated loop is linear with a *linear-step* that is the increment of the associated loop.
- The loop iteration variables in the associated loops of a **simd** construct with multiple associated loops are lastprivate.
- The loop iteration variable in any associated loop of a **loop** construct is lastprivate.
- The implicitly declared variables of a range-based **for** loop are private.
- Variables with static storage duration that are declared in a scope inside the construct are shared.
- If a list item in a **map** clause on the **target** construct has a base pointer, and the base pointer is a scalar variable that does not appear in a **map** clause on the construct, the base pointer is firstprivate.
- If a list item in a **reduction** or **in_reduction** clause on a construct has a base pointer then the base pointer is private.
- Static data members are shared.
- The **__func__** variable and similar function-local predefined variables are shared.

C / C++

Fortran

- Variables declared within a **BLOCK** construct inside a construct that do not have the **SAVE** attribute are private.
- Variables and common blocks that appear in **threadprivate** directives are threadprivate.
- The loop iteration variable in any associated *do-loop* of a **do**, **parallel do**, **taskloop**, or **distribute** construct is private.
- The loop iteration variable in the associated *do-loop* of a **simd** construct with just one associated *do-loop* is linear with a *linear-step* that is the increment of the associated *do-loop*.
- The loop iteration variables in the associated *do-loops* of a **simd** construct with multiple associated *do-loops* are lastprivate.
- The loop iteration variable in any associated *do-loop* of a **loop** construct is lastprivate.
- Loop iteration variables inside **parallel** or task generating constructs are private in the innermost such construct that encloses the loop.
- Implied-do, **FORALL** and **DO CONCURRENT** indices are private.
- Cray pointees have the same data-sharing attribute as the storage with which their Cray pointers are associated. Cray pointer support has been deprecated.
- Assumed-size arrays are shared.
- *Named constants* are shared.
- An associate name that may appear in a variable definition context is shared if its association occurs outside of the construct and otherwise it has the same data-sharing attribute as the selector with which it is associated.

Fortran

Variables with predetermined data-sharing attributes may not be listed in data-sharing attribute clauses, except for the cases listed below. For these exceptions only, listing a predetermined variable in a data-sharing attribute clause is allowed and overrides the variable's predetermined data-sharing attributes.

C / C++

- The loop iteration variable in any associated loop of a **for**, **taskloop**, **distribute**, or **loop** construct may be listed in a **private** or **lastprivate** clause.
- If a **simd** construct has just one associated loop then its loop iteration variable may be listed in a **private**, **lastprivate**, or **linear** clause with a *linear-step* that is the increment of the associated loop.
- If a **simd** construct has more than one associated loop then their loop iteration variables may be listed in a **private** or **lastprivate** clause.

- Variables with **`const`**-qualified type with no mutable members may be listed in a **`firstprivate`** clause, even if they are static data members.
- The **`__func__`** variable and similar function-local predefined variables may be listed in a **`shared`** or **`firstprivate`** clause.

C / C++

Fortran

- The loop iteration variable in any associated *do-loop* of a **`do`**, **`taskloop`**, **`distribute`**, or **`loop`** construct may be listed in a **`private`** or **`lastprivate`** clause.
- The loop iteration variable in the associated *do-loop* of a **`simd`** construct with just one associated *do-loop* may be listed in a **`private`**, **`lastprivate`**, or **`linear`** clause with a *linear-step* that is the increment of the associated loop.
- The loop iteration variables in the associated *do-loops* of a **`simd`** construct with multiple associated *do-loops* may be listed in a **`private`** or **`lastprivate`** clause.
- Loop iteration variables of loops that are not associated with any OpenMP directive may be listed in data-sharing attribute clauses on the surrounding **`teams`**, **`parallel`** or task generating construct, and on enclosed constructs, subject to other restrictions.
- Assumed-size arrays may be listed in a **`shared`** clause.
- *Named constants* may be listed in a **`firstprivate`** clause.

Fortran

Additional restrictions on the variables that may appear in individual clauses are described with each clause in Section 5.4.

Variables with *explicitly determined* data-sharing attributes are those that are referenced in a given construct and are listed in a data-sharing attribute clause on the construct.

Variables with *implicitly determined* data-sharing attributes are those that are referenced in a given construct, do not have predetermined data-sharing attributes, and are not listed in a data-sharing attribute clause on the construct.

Rules for variables with *implicitly determined* data-sharing attributes are as follows:

- In a **`parallel`**, **`teams`**, or task generating construct, the data-sharing attributes of these variables are determined by the **`default`** clause, if present (see Section 5.4.1).
- In a **`parallel`** construct, if no **`default`** clause is present, these variables are shared.
- For constructs other than task generating constructs, if no **`default`** clause is present, these variables reference the variables with the same names that exist in the enclosing context.
- In a **`target`** construct, variables that are not mapped after applying data-mapping attribute rules (see Section 5.8) are firstprivate.

C++

- In an orphaned task generating construct, if no **default** clause is present, formal arguments passed by reference are firstprivate.

C++

Fortran

- In an orphaned task generating construct, if no **default** clause is present, dummy arguments are firstprivate.

Fortran

- In a task generating construct, if no **default** clause is present, a variable for which the data-sharing attribute is not determined by the rules above and that in the enclosing context is determined to be shared by all implicit tasks bound to the current team is shared.
- In a task generating construct, if no **default** clause is present, a variable for which the data-sharing attribute is not determined by the rules above is firstprivate.

Additional restrictions on the variables for which data-sharing attributes cannot be implicitly determined in a task generating construct are described in Section 5.4.4.

5.1.2 Variables Referenced in a Region but not in a Construct

The data-sharing attributes of variables that are referenced in a region, but not in a construct, are determined as follows:

C / C++

- Variables with static storage duration that are declared in called routines in the region are shared.
- File-scope or namespace-scope variables referenced in called routines in the region are shared unless they appear in a **threadprivate** directive.
- Objects with dynamic storage duration are shared.
- Static data members are shared unless they appear in a **threadprivate** directive.
- In C++, formal arguments of called routines in the region that are passed by reference have the same data-sharing attributes as the associated actual arguments.
- Other variables declared in called routines in the region are private.

C / C++

Fortran

- Local variables declared in called routines in the region and that have the **`save`** attribute, or that are data initialized, are shared unless they appear in a **`threadprivate`** directive.
- Variables belonging to common blocks, or accessed by host or use association, and referenced in called routines in the region are shared unless they appear in a **`threadprivate`** directive.
- Dummy arguments of called routines in the region that have the **`VALUE`** attribute are private.
- Dummy arguments of called routines in the region that do not have the **`VALUE`** attribute are private if the associated actual argument is not shared.
- Dummy arguments of called routines in the region that do not have the **`VALUE`** attribute are shared if the actual argument is shared and it is a scalar variable, structure, an array that is not a pointer or assumed-shape array, or a simply contiguous array section. Otherwise, the data-sharing attribute of the dummy argument is implementation-defined if the associated actual argument is shared.
- Cray pointees have the same data-sharing attribute as the storage with which their Cray pointers are associated. Cray pointer support has been deprecated.
- Implied-do indices, **`DO CONCURRENT`** indices, **`FORALL`** indices, and other local variables declared in called routines in the region are private.

Fortran

5.2 `threadprivate` Directive

Name: **`threadprivate`** **Category:** declarative	**Association:** none **Properties:** default

Arguments: **`threadprivate`** **(***list***)**

Name	Type	Properties
list	List containing variable list item	default

Semantics

The **`threadprivate`** directive specifies that variables are replicated, with each thread having its own copy.

Unless otherwise specified, each copy of a threadprivate variable is initialized once, in the manner specified by the program, but at an unspecified point in the program prior to the first reference to that copy. The storage of all copies of a threadprivate variable is freed according to how static variables are handled in the base language, but at an unspecified point in the program.

C++

Each copy of a block-scope threadprivate variable that has a dynamic initializer is initialized the first time its thread encounters its definition; if its thread does not encounter its definition, its initialization is unspecified.

C++

The content of a threadprivate variable can change across a task scheduling point if the executing thread switches to another task that modifies the variable. For more details on task scheduling, see Section 1.3 and Section 12.

In **parallel** regions, references by the primary thread will be to the copy of the variable in the thread that encountered the **parallel** region.

During a sequential part references will be to the initial thread's copy of the variable. The values of data in the initial thread's copy of a threadprivate variable are guaranteed to persist between any two consecutive references to the variable in the program provided that no **teams** construct that is not nested inside of a **target** construct is encountered between the references and that the initial thread is not nested inside of a **teams** region. For initial threads nested inside of a **teams** region, the values of data in the copies of a threadprivate variable of those initial threads are guaranteed to persist between any two consecutive references to the variable inside that **teams** region.

The values of data in the threadprivate variables of threads that are not initial threads are guaranteed to persist between two consecutive active **parallel** regions only if all of the following conditions hold:

- Neither **parallel** region is nested inside another explicit **parallel** region;
- The number of threads used to execute both **parallel** regions is the same;
- The thread affinity policies used to execute both **parallel** regions are the same;
- The value of the *dyn-var* internal control variable in the enclosing task region is *false* at entry to both **parallel** regions;
- No **teams** construct that is not nested inside of a **target** construct is encountered between the **parallel** regions;
- No construct with an **order** clause that specifies **concurrent** is encountered between the **parallel** regions; and
- Neither the **omp_pause_resource** nor **omp_pause_resource_all** routine is called.

If these conditions all hold, and if a threadprivate variable is referenced in both regions, then threads with the same thread number in their respective regions will reference the same copy of that variable.

C / C++

If the above conditions hold, the storage duration, lifetime, and value of a thread's copy of a threadprivate variable that does not appear in any **copyin** clause on the second region will span the two consecutive active **parallel** regions. Otherwise, the storage duration, lifetime, and value of a thread's copy of the variable in the second region is unspecified.

C / C++

Fortran

If the above conditions hold, the definition, association, or allocation status of a thread's copy of a threadprivate variable or a variable in a threadprivate common block that is not affected by any **copyin** clause that appears on the second region (a variable is affected by a **copyin** clause if the variable appears in the **copyin** clause or it is in a common block that appears in the **copyin** clause) will span the two consecutive active **parallel** regions. Otherwise, the definition and association status of a thread's copy of the variable in the second region are undefined, and the allocation status of an allocatable variable will be implementation defined.

If a threadprivate variable or a variable in a threadprivate common block is not affected by any **copyin** clause that appears on the first **parallel** region in which it is referenced, the thread's copy of the variable inherits the declared type parameter and the default parameter values from the original variable. The variable or any subobject of the variable is initially defined or undefined according to the following rules:

- If it has the **ALLOCATABLE** attribute, each copy created will have an initial allocation status of unallocated;
- If it has the **POINTER** attribute, each copy will have the same association status as the initial association status.
- If it does not have either the **POINTER** or the **ALLOCATABLE** attribute:
  - If it is initially defined, either through explicit initialization or default initialization, each copy created is so defined;
  - Otherwise, each copy created is undefined.

Fortran

C++

The order in which any constructors for different threadprivate variables of class type are called is unspecified. The order in which any destructors for different threadprivate variables of class type are called is unspecified.

C++

Restrictions

Restrictions to the **`threadprivate`** directive are as follows:

- A thread must not reference another thread's copy of a threadprivate variable.
- A threadprivate variable must not be used in a list item in any clause except for the **`copyin`** and **`copyprivate`** clauses.
- A program in which an untied task accesses threadprivate storage is non-conforming.

C / C++

- Each *list-item* must be a file-scope, namespace-scope, or static block-scope variable.
- No *list-item* may have an incomplete type.
- The address of a threadprivate variable may not be an address constant.
- If the value of a variable referenced in an explicit initializer of a threadprivate variable is modified prior to the first reference to any instance of the threadprivate variable, then the behavior is unspecified.
- A variable that is part of another variable (as an array element or a structure element) cannot appear in a **`threadprivate`** directive unless it is a static data member of a C++ class.
- A **`threadprivate`** directive for file-scope variables must appear outside any definition or declaration, and must lexically precede all references to any of the variables in its list.
- A **`threadprivate`** directive for namespace-scope variables must appear outside any definition or declaration other than the namespace definition itself, and must lexically precede all references to any of the variables in its list.
- Each variable in the list of a **`threadprivate`** directive at file, namespace, or class scope must refer to a variable declaration at file, namespace, or class scope that lexically precedes the directive.
- A **`threadprivate`** directive for static block-scope variables must appear in the scope of the variable and not in a nested scope. The directive must lexically precede all references to any of the variables in its list.
- Each variable in the list of a **`threadprivate`** directive in block scope must refer to a variable declaration in the same scope that lexically precedes the directive. The variable must have static storage duration.
- If a variable is specified in a **`threadprivate`** directive in one translation unit, it must be specified in a **`threadprivate`** directive in every translation unit in which it is declared.

C / C++

C++

- A **threadprivate** directive for static class member variables must appear in the class definition, in the same scope in which the member variables are declared, and must lexically precede all references to any of the variables in its list.
- A threadprivate variable must not have an incomplete type or a reference type.
- A threadprivate variable with class type must have:
  - An accessible, unambiguous default constructor in the case of default initialization without a given initializer;
  - An accessible, unambiguous constructor that accepts the given argument in the case of direct initialization; and
  - An accessible, unambiguous copy constructor in the case of copy initialization with an explicit initializer.

C++

Fortran

- Each *list-item* must be a named variable or a named common block; a named common block must appear between slashes.
- A coarray cannot appear in a **threadprivate** directive.
- An associate name cannot appear in a **threadprivate** directive.
- The **threadprivate** directive must appear in the declaration section of a scoping unit in which the common block or variable is declared.
- If a **threadprivate** directive that specifies a common block name appears in one program unit, then such a directive must also appear in every other program unit that contains a **COMMON** statement that specifies the same name. It must appear after the last such **COMMON** statement in the program unit.
- If a threadprivate variable or a threadprivate common block is declared with the **BIND** attribute, the corresponding C entities must also be specified in a **threadprivate** directive in the C program.
- A variable can only appear in a **threadprivate** directive in the scope in which it is declared. It must not be an element of a common block or appear in an **EQUIVALENCE** statement.
- A variable that appears in a **threadprivate** directive must be declared in the scope of a module or have the **SAVE** attribute, either explicitly or implicitly.
- The effect of an access to a threadprivate variable in a **DO CONCURRENT** construct is unspecified.

Fortran

Cross References

- *dyn-var* ICV, see Section 2.
- Number of threads used to execute a **parallel** region, see Section 10.1.1.
- **order** clause, see Section 10.3.
- **copyin** clause, see Section 5.7.1.

5.3 List Item Privatization

Some data-sharing attribute clauses, including reduction clauses, specify that list item that appear in their *list* parameter may be privatized for the construct on which they appear. Each task that references a privatized list item in any statement in the construct receives at least one new list item if the construct has one or more associated loops, and otherwise each such task receives one new list item. Each SIMD lane used in a **simd** construct that references a privatized list item in any statement in the construct receives at least one new list item. Language-specific attributes for new list items are derived from the corresponding original list item. Inside the construct, all references to the original list item are replaced by references to a new list item received by the task or SIMD lane.

If the construct has one or more associated loops, within the same logical iteration of the loops, then the same new list item replaces all references to the original list item. For any two logical iterations, if the references to the original list item are replaced by the same list item then the logical iterations must execute in some sequential order.

In the rest of the region, whether references are to a new list item or the original list item is unspecified. Therefore, if an attempt is made to reference the original item, its value after the region is also unspecified. If a task or a SIMD lane does not reference a privatized list item, whether the task or SIMD lane receives a new list item is unspecified.

The value and/or allocation status of the original list item will change only:

- If accessed and modified via pointer;
- If possibly accessed in the region but outside of the construct;
- As a side effect of directives or clauses; or

Fortran

- If accessed and modified via construct association.

Fortran

C++

If the construct is contained in a member function, whether accesses anywhere in the region through the implicit **this** pointer refer to the new list item or the original list item is unspecified.

C++

C / C++

A new list item of the same type, with automatic storage duration, is allocated for the construct. The storage and thus lifetime of these list items last until the block in which they are created exits. The size and alignment of the new list item are determined by the type of the variable. This allocation occurs once for each task generated by the construct and once for each SIMD lane used by the construct.

The new list item is initialized, or has an undefined initial value, as if it had been locally declared without an initializer.

C / C++

C++

If the type of a list item is a reference to a type *T* then the type will be considered to be *T* for all purposes of this clause.

The order in which any default constructors for different private variables of class type are called is unspecified. The order in which any destructors for different private variables of class type are called is unspecified.

C++

Fortran

If any statement of the construct references a list item, a new list item of the same type and type parameters is allocated. This allocation occurs once for each task generated by the construct and once for each SIMD lane used by the construct. If the type of the list item has default initialization, the new list item has default initialization. Otherwise, the initial value of the new list item is undefined. The initial status of a private pointer is undefined.

For a list item or the subobject of a list item with the **ALLOCATABLE** attribute:

- If the allocation status is unallocated, the new list item or the subobject of the new list item will have an initial allocation status of unallocated;
- If the allocation status is allocated, the new list item or the subobject of the new list item will have an initial allocation status of allocated; and
- If the new list item or the subobject of the new list item is an array, its bounds will be the same as those of the original list item or the subobject of the original list item.

A privatized list item may be storage-associated with other variables when the data-sharing attribute clause is encountered. Storage association may exist because of constructs such as **EQUIVALENCE** or **COMMON**. If *A* is a variable that is privatized by a construct and *B* is a variable that is storage-associated with *A*, then:

- The contents, allocation, and association status of *B* are undefined on entry to the region;
- Any definition of *A*, or of its allocation or association status, causes the contents, allocation, and association status of *B* to become undefined; and

- Any definition of *B*, or of its allocation or association status, causes the contents, allocation, and association status of *A* to become undefined.

A privatized list item may be a selector of an **ASSOCIATE** or **SELECT TYPE** construct. If the construct association is established prior to a **parallel** region, the association between the associate name and the original list item will be retained in the region.

Finalization of a list item of a finalizable type or subobjects of a list item of a finalizable type occurs at the end of the region. The order in which any final subroutines for different variables of a finalizable type are called is unspecified.

Fortran

If a list item appears in both **firstprivate** and **lastprivate** clauses, the update required for the **lastprivate** clause occurs after all initializations for the **firstprivate** clause.

Restrictions

The following restrictions apply to any list item that is privatized unless otherwise stated for a given data-sharing attribute clause:

C++

- A variable of class type (or array thereof) that is privatized requires an accessible, unambiguous default constructor for the class type.

C++

C / C++

- A variable that is privatized must not have a **const**-qualified type unless it is of class type with a **mutable** member. This restriction does not apply to the **firstprivate** clause.
- A variable that is privatized must not have an incomplete type or be a reference to an incomplete type.

C / C++

Fortran

- Variables that appear in namelist statements, in variable format expressions, and in expressions for statement function definitions, may not be privatized.
- Pointers with the **INTENT(IN)** attribute may not be privatized. This restriction does not apply to the **firstprivate** clause.
- A private variable must not be coindexed or appear as an actual argument to a procedure where the corresponding dummy argument is a coarray.
- Assumed-size arrays may not be privatized in a **target**, **teams**, or **distribute** construct.

Fortran

5.4 Data-Sharing Attribute Clauses

Several constructs accept clauses that allow a user to control the data-sharing attributes of variables referenced in the construct. Not all of the clauses listed in this section are valid on all directives. The set of clauses that is valid on a particular directive is described with the directive.

All list items that appear in a data-sharing attribute clause must be visible, according to the scoping rules of the base language. A list item may not appear in more than one clause on the same directive, except that it may be specified in both **`firstprivate`** and **`lastprivate`** clauses.

The reduction data-sharing attribute clauses are explained in Section 5.5.

C++

If a variable referenced in a data-sharing attribute clause has a type derived from a template, and the program does not otherwise reference that variable then any behavior related to that variable is unspecified.

C++

Fortran

If individual members of a common block appear in a data-sharing attribute clause other than the **`shared`** clause, the variables no longer have a Fortran storage association with the common block.

Fortran

5.4.1 `default` Clause

Name:	**Properties:**
`default`	unique

Arguments:

Name	Type	Properties
data-sharing-attribute	**Keyword:** **`firstprivate`**, **`none`**, **`private`**, **`shared`**	default

Directives:

`parallel`, **`task`**, **`taskloop`**, **`teams`**

Semantics

The **`default`** clause explicitly determines the data-sharing attributes of variables that are referenced in construct and would otherwise be implicitly determined (see Section 5.1.1).

If *data-sharing-attribute* is **`shared`** or, for Fortran, **`firstprivate`** or **`private`**, the data-sharing attribute of all variables referenced in the construct that have implicitly determined data-sharing attributes will be *data-sharing-attribute*.

C / C++

If *data-sharing-attribute* is **`firstprivate`** or **`private`**, each variable with static storage duration that is declared in a namespace or global scope and referenced in the construct, and that does not have a predetermined data-sharing attribute, must have its data-sharing attribute explicitly determined by being listed in a data-sharing attribute clause. The data-sharing attribute of all other variables that are referenced in the construct and that have implicitly determined data-sharing attributes will be *data-sharing-attribute*.

C / C++

The **`default(none)`** clause requires that each variable that is referenced in the construct, and that does not have a predetermined data-sharing attribute, must have its data-sharing attribute explicitly determined by being listed in a data-sharing attribute clause.

5.4.2 `shared` Clause

Name:	**Properties:**
`shared`	data-environment attribute, data-sharing attribute

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Directives:

`parallel`, **`task`**, **`taskloop`**, **`teams`**

Semantics

The **`shared`** clause declares one or more list items to be shared by tasks generated by the construct on which it appears. All references to a list item within a task refer to the storage area of the original variable at the point the directive was encountered.

The programmer must ensure, by adding proper synchronization, that storage shared by an explicit task region does not reach the end of its lifetime before the explicit task region completes its execution.

Fortran

The association status of a shared pointer becomes undefined upon entry to and exit from the construct if it is associated with a target or a subobject of a target that appears as a privatized list item in a data-sharing attribute clause on the construct. A reference to the shared storage that is associated with the dummy argument by any other task must be synchronized with the reference to the procedure to avoid possible data races.

Fortran

5.4.3 `private` Clause

Name:	Properties:
`private`	data-environment attribute, data-sharing attribute, privatization

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Directives:

`distribute`, **`do`**, **`for`**, **`loop`**, **`parallel`**, **`scope`**, **`sections`**, **`simd`**, **`single`**, **`target`**, **`task`**, **`taskloop`**, **`teams`**

Semantics

The **`private`** clause specifies that its list items are to be privatized according to Section 5.3. Each task or SIMD lane that references a list item in the construct receives only one new list item, unless the construct has one or more associated loops and an **`order`** clause that specifies **`concurrent`** is also present.

Restrictions

Restrictions to the **`private`** clause are as specified in Section 5.3.

Cross References

- List Item Privatization, see Section 5.3.

5.4.4 `firstprivate` Clause

Name:	Properties:
`firstprivate`	data-environment attribute, data-sharing attribute, privatization

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Directives:
distribute, **do**, **for**, **parallel**, **scope**, **sections**, **single**, **target**, **task**, **taskloop**, **teams**

Semantics

The **firstprivate** clause provides a superset of the functionality provided by the **private** clause. A list item that appears in a **firstprivate** clause is subjectto the **private** clause semantics described in Section 5.4.3, except as noted. In addition, the new list item is initialized from the original list item that exists before the construct. The initialization of the new list item is done once for each task that references the list item in any statement in the construct. The initialization is done prior to the execution of the construct.

For a **firstprivate** clause on a **parallel**, **task**, **taskloop**, **target**, or **teams** construct, the initial value of the new list item is the value of the original list item that exists immediately prior to the construct in the task region where the construct is encountered unless otherwise specified. For a **firstprivate** clause on a worksharing construct, the initial value of the new list item for each implicit task of the threads that execute the worksharing construct is the value of the original list item that exists in the implicit task immediately prior to the point in time that the worksharing construct is encountered unless otherwise specified.

To avoid data races, concurrent updates of the original list item must be synchronized with the read of the original list item that occurs as a result of the **firstprivate** clause.

C / C++

For variables of non-array type, the initialization occurs by copy assignment. For an array of elements of non-array type, each element is initialized as if by assignment from an element of the original array to the corresponding element of the new array.

C / C++

C++

For each variable of class type:

- If the **firstprivate** clause is not on a **target** construct then a copy constructor is invoked to perform the initialization; and
- If the **firstprivate** clause is on a **target** construct then how many copy constructors, if any, are invoked is unspecified.

If copy constructors are called, the order in which copy constructors for different variables of class type are called is unspecified.

C++

Fortran

If the original list item does not have the **POINTER** attribute, initialization of the new list items occurs as if by intrinsic assignment unless the original list item has a compatible type-bound defined assignment, in which case initialization of the new list items occurs as if by the defined assignment. If the original list item that does not have the **POINTER** attribute has the allocation status of unallocated, the new list items will have the same status.

If the original list item has the **POINTER** attribute, the new list items receive the same association status of the original list item as if by pointer assignment.

The list items that appear in a **firstprivate** clause may include *named constants*.

Fortran

Restrictions

Restrictions to the **firstprivate** clause are as follows:

- A list item that is private within a **parallel** region must not appear in a **firstprivate** clause on a worksharing construct if any of the worksharing regions that arise from the worksharing construct ever bind to any of the **parallel** regions that arise from the **parallel** construct.
- A list item that is private within a **teams** region must not appear in a **firstprivate** clause on a **distribute** construct if any of the **distribute** regions that arise from the **distribute** construct ever bind to any of the **teams** regions that arise from the **teams** construct.
- A list item that appears in a **reduction** clause of a **parallel** construct must not appear in a **firstprivate** clause on a worksharing, **task**, or **taskloop** construct if any of the worksharing or **task** regions that arise from the worksharing, **task**, or **taskloop** construct ever bind to any of the **parallel** regions that arise from the **parallel** construct.
- A list item that appears in a **reduction** clause of a **teams** construct must not appear in a **firstprivate** clause on a **distribute** construct if any of the **distribute** regions that arise from the **distribute** construct ever bind to any of the **teams** regions that arise from the **teams** construct.
- A list item that appears in a **reduction** clause of a worksharing construct must not appear in a **firstprivate** clause in a **task** construct encountered during execution of any of the worksharing regions that arise from the worksharing construct.

C++

- A variable of class type (or array thereof) that appears in a **firstprivate** clause requires an accessible, unambiguous copy constructor for the class type.

C++

C / C++

- If a list item in a **firstprivate** clause on a worksharing construct has a reference type then it must bind to the same object for all threads of the team.

C / C++

Fortran

- If the list item is a polymorphic variable with the **ALLOCATABLE** attribute, the behavior is unspecified.

Fortran

Cross References

- **private** clause, see Section 5.4.3.

5.4.5 lastprivate Clause

Name:
lastprivate

Properties:
data-environment attribute, data-sharing attribute, privatization

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Modifiers:

Name	Modifies	Type	Properties
lastprivate-modifier	*list*	**Keyword: conditional**	default

Directives:

distribute, **do**, **for**, **loop**, **sections**, **simd**, **taskloop**

Semantics

The **lastprivate** clause provides a superset of the functionality provided by the **private** clause. A list item that appears in a **lastprivate** clause is subject to the **private** clause semantics described in Section 5.4.3. In addition, when a **lastprivate** clause without the **conditional** modifier appears on a directive and the list item is not an iteration variable of one of the associated loops, the value of each new list item from the sequentially last iteration of the associated loops, or the lexically last **section** construct, is assigned to the original list item. When the **conditional** modifier appears on the clause or the list item is an iteration variable of one of the associated loops, if sequential execution of the loop nest would assign a value to the list item then the original list item is assigned the value that the list item would have after sequential execution of the loop nest.

C / C++

For an array of elements of non-array type, each element is assigned to the corresponding element of the original array.

C / C++

Fortran

If the original list item does not have the **POINTER** attribute, its update occurs as if by intrinsic assignment unless it has a type bound procedure as a defined assignment.

If the original list item has the **POINTER** attribute, its update occurs as if by pointer assignment.

Fortran

When the **conditional** modifier does not appear on the **lastprivate** clause, any list item that is not an iteration variable of the associated loops and that is not assigned a value by the sequentially last iteration of the loops, or by the lexically last **section** construct, has an unspecified value after the construct. When the **conditional** modifier does not appear on the **lastprivate** clause, a list item that is the iteration variable of an associated loop and that would not be assigned a value during sequential execution of the loop nest has an unspecified value after the construct. Unassigned subcomponents also have unspecified values after the construct.

If the **lastprivate** clause is used on a construct to which neither the **nowait** nor the **nogroup** clauses are applied, the original list item becomes defined at the end of the construct. To avoid data races, concurrent reads or updates of the original list item must be synchronized with the update of the original list item that occurs as a result of the **lastprivate** clause.

Otherwise, If the **lastprivate** clause is used on a construct to which the **nowait** or the **nogroup** clauses are applied, accesses to the original list item may create a data race. To avoid this data race, if an assignment to the original list item occurs then synchronization must be inserted to ensure that the assignment completes and the original list item is flushed to memory.

If a list item that appears in a **lastprivate** clause with the **conditional** modifier is modified in the region by an assignment outside the construct or not to the list item then the value assigned to the original list item is unspecified.

Restrictions

Restrictions to the **lastprivate** clause are as follows:

- A list item that is private within a **parallel** region, or that appears in the **reduction** clause of a **parallel** construct, must not appear in a **lastprivate** clause on a worksharing construct if any of the corresponding worksharing regions ever binds to any of the corresponding **parallel** regions.
- A list item that appears in a **lastprivate** clause with the **conditional** modifier must be a scalar variable.

C++

- A variable of class type (or array thereof) that appears in a **lastprivate** clause requires an accessible, unambiguous default constructor for the class type, unless the list item is also specified in a **firstprivate** clause.
- A variable of class type (or array thereof) that appears in a **lastprivate** clause requires an accessible, unambiguous copy assignment operator for the class type. The order in which copy assignment operators for different variables of class type are called is unspecified.
- If a list item in a **lastprivate** clause on a worksharing construct has a reference type then it must bind to the same object for all threads of the team.

C++

Fortran

- A variable that appears in a **lastprivate** clause must be definable.
- If the original list item has the **ALLOCATABLE** attribute, the corresponding list item of which the value is assigned to the original item must have an allocation status of allocated upon exit from the sequentially last iteration or lexically last **section** construct.
- If the list item is a polymorphic variable with the **ALLOCATABLE** attribute, the behavior is unspecified.

Fortran

Cross References

- **private** clause, see Section 5.4.3.

5.4.6 linear Clause

Name:	**Properties:**
linear	data-environment attribute, data-sharing attribute, privatization, post-modified

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Modifiers:

Name	Modifies	Type	Properties
linear-step	*list*	OpenMP integer expression	unique, ultimate, region-invariant
linear-modifier	*list*	**Keyword: ref, uval, val**	unique

Directives:

declare simd, **do**, **for**, **simd**

Additional information: *list* and *linear-modifier* may instead be specified as *linear-modifier* **(***list***)** for **linear** clauses that appear on a **declare simd** directive. This syntax has been deprecated.

Semantics

The **linear** clause provides a superset of the functionality provided by the **private** clause. A list item that appears in a **linear** clause is subject to the **private** clause semantics described in Section 5.4.3 except as noted. If *linear-step* is not specified, it is assumed to be 1. If *linear-modifier* is not specified, the effect is as if the **val** modifier is specified.

When a **linear** clause is specified on a construct, the value of the new list item on each logical iteration of the associated loops corresponds to the value of the original list item before entering the construct plus the logical number of the iteration times *linear-step*. The value corresponding to the sequentially last logical iteration of the associated loops is assigned to the original list item.

When a **linear** clause is specified on a **declare simd** directive, the list items refer to parameters of the procedure to which the directive applies. For a given call to the procedure, the clause determines whether the SIMD version generated by the directive may be called. If the clause does not specify the **ref** *linear-modifier*, the SIMD version requires that the value of the corresponding argument at the callsite is equal to the value of the argument from the first lane plus the logical number of the lane times the *linear-step*. If the clause specifies the **ref** *linear-modifier*, the SIMD version requires that the storage locations of the corresponding arguments at the callsite from each SIMD lane correspond to locations within a hypothetical array of elements of the same type, indexed by the logical number of the lane times the *linear-step*.

Restrictions

Restrictions to the **linear** clause are as follows:

- Only a loop iteration variable of a loop that is associated with the construct may appear as a *list-item* in a **linear** clause if a **reduction** clause with the **inscan** modifier also appears on the construct.
- A *linear-modifier* may be specified as **ref** or **uval** only on a **declare simd** directive.
- For a **linear** clause that appears on a loop-associated construct, the difference between the value of a list item at the end of a logical iteration and its value at the beginning of the logical iteration must be equal to *linear-step*.
- If *linear-modifier* is **uval** for a list item in a **linear** clause that is specified on a **declare simd** directive and the list item is modified during a call to the SIMD version of the procedure, the program must not depend on the value of the list item upon return from the procedure.
- If *linear-modifier* is **uval** for a list item in a **linear** clause that is specified on a **declare simd** directive, the program must not depend on the storage of the argument in the procedure being the same as the storage of the corresponding argument at the callsite.

C

- All list items must be of integral or pointer type.
- If specified, *linear-modifier* must be **val**.

C

C++

- If *linear-modifier* is not **ref**, all list items must be of integral or pointer type, or must be a reference to an integral or pointer type.
- If *linear-modifier* is **ref** or **uval**, all list items must be of a reference type.
- If a list item in a **linear** clause on a worksharing construct has a reference type then it must bind to the same object for all threads of the team.
- If a list item in a **linear** clause that is specified on a **declare simd** directive is of a reference type and *linear-modifier* is not **ref**, the difference between the value of the argument on exit from the function and its value on entry to the function must be the same for all SIMD lanes.

C++

Fortran

- If *linear-modifier* is not **ref** all list items must be of type **integer**.
- If *linear-modifier* is **ref** or **uval**, all list items must be dummy arguments without the **VALUE** attribute.
- List items must not be Cray pointers or variables that have the **POINTER** attribute. Cray pointer support has been deprecated.
- If *linear-modifier* is not **ref** and a list item has the **ALLOCATABLE** attribute, the allocation status of the list item in the sequentially last iteration must be allocated upon exit from that iteration.
- If *linear-modifier* is **ref**, list items must be polymorphic variables, assumed-shape arrays, or variables with the **ALLOCATABLE** attribute.
- If a list item in a **linear** clause that is specified on a **declare simd** directive is a dummy argument without the **VALUE** attribute and *linear-modifier* is not **ref**, the difference between the value of the argument on exit from the procedure and its value on entry to the procedure must be the same for all SIMD lanes.
- A common block name cannot appear in a **linear** clause.

Fortran

Cross References

- Worksharing-loop constructs, see Section 11.5.
- **declare simd** directive, see Section 7.7.
- **simd** construct, see Section 10.4.
- **taskloop** construct, see Section 12.6.
- **private** clause, see Section 5.4.3.

5.4.7 `uniform` Clause

Name:	Properties:
uniform	data-environment attribute, data-sharing attribute, privatization

Arguments:

Name	Type	Properties
parameter-list	List containing parameter list item	default

Directives:

declare simd

Semantics

The **uniform** clause declares one or more arguments to have an invariant value for all concurrent invocations of the function in the execution of a single SIMD loop.

Cross References

- **declare simd** directive, see Section 7.7.
- **private** clause, see Section 5.4.3.

5.5 Reduction Clauses and Directives

The reduction clauses are data-sharing attribute clauses that can be used to perform some forms of recurrence calculations in parallel. Reduction clauses include reduction scoping clauses and reduction participating clauses. Reduction scoping clauses define the region in which a reduction is computed. Reduction participating clauses define the participants in the reduction.

5.5.1 OpenMP Reduction Identifiers

The syntax of an OpenMP reduction identifier is defined as follows:

C

A reduction identifier is either an *identifier* or one of the following operators: **+**, **-** (deprecated), *****, **&**, **|**, **^**, **&&** and **||**.

C

C++

A reduction identifier is either an *id-expression* or one of the following operators: **+**, **-** (deprecated), *****, **&**, **|**, **^**, **&&** and **||**.

C++

Fortran

A reduction identifier is either a base language identifier, or a user-defined operator, or one of the following operators: **+**, **-** (deprecated), *****, **.and.**, **.or.**, **.eqv.**, **.neqv.**, or one of the following intrinsic procedure names: **max**, **min**, **iand**, **ior**, **ieor**.

Fortran

5.5.2 OpenMP Combiner Expressions

A *combiner expression* specifies how a reduction combines partial results into a single value.

Fortran

A combiner expression is an assignment statement or a subroutine name followed by an argument list.

Fortran

In the definition of a combiner expression, **omp_in** and **omp_out** correspond to two special variable identifiers that refer to storage of the type of the reduction list item to which the reduction applies. If the list item is an array or array section, the identifiers to which **omp_in** and **omp_out** correspond each refer to an array element. Each of the two special variable identifiers denotes one of the values to be combined before executing the combiner expression. The special **omp_out** identifier refers to the storage that holds the resulting combined value after executing the combiner expression. The number of times that the combiner expression is executed and the order of these executions for any reduction clause are unspecified.

Fortran

If the combiner expression is a subroutine name with an argument list, the combiner expression is evaluated by calling the subroutine with the specified argument list. If the combiner expression is an assignment statement, the combiner expression is evaluated by executing the assignment statement.

If a generic name is used in a combiner expression and the list item in the corresponding reduction clause is an array or array section, it is resolved to the specific procedure that is elemental or only has scalar dummy arguments.

Fortran

Restrictions

Restrictions to combiner expressions are as follows:

- The only variables allowed in a combiner expression are **omp_in** and **omp_out**.
- If execution of a combiner expression results in the execution of an OpenMP construct or an OpenMP API call, the behavior is unspecified.

C / C++

- If a combiner expression corresponds to a reduction identifier that is used in a **target** region, a declare target directive must be specified for any function that can be accessed through the expression.

C / C++

Fortran

- Any selectors in the designator of **omp_in** and **omp_out** must be *component selectors*.
- Any subroutine or function used in a combiner expression must be an intrinsic function, or must have an accessible interface.
- Any user-defined operator, defined assignment or extended operator used in a combiner expression must have an accessible interface.
- If any subroutine, function, user-defined operator, defined assignment or extended operator is used in a combiner expression, it must be accessible to the subprogram in which the corresponding **reduction** clause is specified.
- Any subroutine used in a combiner expression must not have any alternate returns appear in the argument list.
- If the list item in the corresponding **reduction** clause is an array or array section, any procedure used in a combiner expression must either be elemental or have dummy arguments that are scalar.
- Any procedure called in the region of a combiner expression must be pure and may not reference any host-associated variables.
- If a combiner expression corresponds to a reduction identifier that is used in a **target** region, a **declare target** directive must be specified for any function or subroutine that can be accessed through the expression.

Fortran

5.5.3 OpenMP Initializer Expressions

An *initializer expression* determines the initializer for the private copies of reduction list items. If the initialization of the copies is not determined *a priori*, the syntax of an initializer expression is as follows:

C

```
omp_priv = initializer
```

C

or

C++

```
omp_priv initializer
```

C++

or

C / C++

```
function-name (argument-list)
```

C / C++

or

Fortran

```
omp_priv = expression
```

or

```
subroutine-name (argument-list)
```

Fortran

In the definition of an initializer expression, the **omp_priv** special identifier refers to the storage to be initialized. The special identifier **omp_orig** can be used in an initializer expression to refer to the storage of the original variable to be reduced. The number of times that an initializer expression is evaluated and the order of these evaluations are unspecified.

C / C++

If an initializer expression is a function name with an argument list, it is evaluated by calling the function with the specified argument list. Otherwise, an initializer expression specifies how **omp_priv** is declared and initialized.

C / C++

Fortran

If an initializer expression is a subroutine name with an argument list, the *initializer-expr* is evaluated by calling the subroutine with the specified argument list. If an initializer expression is an assignment statement, the initializer expression is evaluated by executing the assignment statement.

Fortran

C

The *a priori* initialization of private copies that are created for reductions follows the rules for initialization of objects with static storage duration.

C

C++

The *a priori* initialization of private copies that are created for reductions follows the rules for *default-initialization*.

C++

Fortran

The rules for *a priori* initialization of private copies that are created for reductions are as follows:

- For **complex**, **real**, or **integer** types, the value 0 will be used.
- For **logical** types, the value **.false.** will be used.
- For derived types for which default initialization is specified, default initialization will be used.
- Otherwise, the behavior is unspecified.

Fortran

Restrictions

Restrictions to initializer expressions are as follows:

- The only variables allowed in an initializer expression are **omp_priv** and **omp_orig**.
- If an initializer expression modifes the variable **omp_orig**, the behavior is unspecified.
- If execution of an initializer expression results in the execution of an OpenMP construct or an OpenMP API call, the behavior is unspecified.

C

- If an initializer expression is a function name with an argument list, one of the arguments must be the address of **omp_priv**.

C

C++

- If an initializer expression is a function name with an argument list, one of the arguments must be **omp_priv** or the address of **omp_priv**.

C++

C / C++

- If an initializer expression corresponds to a reduction identifier that is used in a **target** region, a declare target directive must be specified for any function that can be accessed through the expression.

C / C++

Fortran

- If an initializer expression is a subroutine name with an argument list, one of the arguments must be **omp_priv**.
- Any subroutine or function used in an initializer expression must be an intrinsic function, or must have an accessible interface.
- Any user-defined operator, defined assignment or extended operator used in an initializer expression must have an accessible interface.
- If any subroutine, function, user-defined operator, defined assignment or extended operator is used in an initializer expression, it must be accessible to the subprogram in which the corresponding **reduction** clause is specified.
- Any subroutine used in an initializer expression must not have any alternate returns appear in the argument list.
- If the list item in the corresponding **reduction** clause is an array or array section, any procedure used in the initializer expression must either be elemental or have dummy arguments that are scalar.
- Any procedure called in the region of an initializer expression must be pure and may not reference any host-associated variables.
- If an initializer expression corresponds to a reduction identifier that is used in a **target** region, a **declare target** directive must be specified for any function or subroutine that can be accessed through the expression.

Fortran

5.5.4 Implicitly Declared OpenMP Reduction Identifiers

C / C++

Table 5.1 lists each reduction identifier that is implicitly declared at every scope for arithmetic types and its semantic initializer value. The actual initializer value is that value as expressed in the data type of the reduction list item.

TABLE 5.1: Implicitly Declared C/C++ Reduction Identifiers

Identifier	Initializer	Combiner
`+`	`omp_priv = 0`	`omp_out += omp_in`
`-` (deprecated)	`omp_priv = 0`	`omp_out += omp_in`
`*`	`omp_priv = 1`	`omp_out *= omp_in`
`&`	`omp_priv = ~ 0`	`omp_out &= omp_in`
`|`	`omp_priv = 0`	`omp_out |= omp_in`
`^`	`omp_priv = 0`	`omp_out ^= omp_in`
`&&`	`omp_priv = 1`	`omp_out = omp_in && omp_out`
`||`	`omp_priv = 0`	`omp_out = omp_in || omp_out`
`max`	`omp_priv =` *Minimal representable number in the reduction list item type*	`omp_out = omp_in > omp_out ? omp_in : omp_out`
`min`	`omp_priv =` *Maximal representable number in the reduction list item type*	`omp_out = omp_in < omp_out ? omp_in : omp_out`

C / C++

Fortran

Table 5.2 lists each reduction identifier that is implicitly declared for numeric and logical types and its semantic initializer value. The actual initializer value is that value as expressed in the data type of the reduction list item.

TABLE 5.2: Implicitly Declared Fortran Reduction Identifiers

Identifier	Initializer	Combiner
`+`	`omp_priv = 0`	`omp_out = omp_in + omp_out`
`-` (deprecated)	`omp_priv = 0`	`omp_out = omp_in + omp_out`
`*`	`omp_priv = 1`	`omp_out = omp_in * omp_out`
`.and.`	`omp_priv = .true.`	`omp_out = omp_in .and. omp_out`

table continued on next page

table continued from previous page

Identifier	Initializer	Combiner
`.or.`	`omp_priv = .false.`	`omp_out = omp_in .or. omp_out`
`.eqv.`	`omp_priv = .true.`	`omp_out = omp_in .eqv. omp_out`
`.neqv.`	`omp_priv = .false.`	`omp_out = omp_in .neqv. omp_out`
`max`	`omp_priv =` *Minimal representable number in the reduction list item type*	`omp_out = max(omp_in, omp_out)`
`min`	`omp_priv =` *Maximal representable number in the reduction list item type*	`omp_out = min(omp_in, omp_out)`
`iand`	`omp_priv =` *All bits on*	`omp_out = iand(omp_in, omp_out)`
`ior`	`omp_priv = 0`	`omp_out = ior(omp_in, omp_out)`
`ieor`	`omp_priv = 0`	`omp_out = ieor(omp_in, omp_out)`

Fortran

5.5.5 `initializer` Clause

Name: `initializer`

Properties: unique

Arguments:

Name	Type	Properties
initializer-expr	Expression of type initializer	default

Directives:

`declare reduction`

Semantics

The `initializer` clause can be used to specify *initializer-expr* as the initializer expression for a user-defined reduction.

5.5.6 Properties Common to All Reduction Clauses

A reduction *clause-specification* has an *clause-argument-specification* that specifies an OpenMP variable list argument that has a required modifier that specifies the reduction identifier to be performed for the reduction. The reduction identifier must match a previously declared reduction identifier of the same name and type for each of the list items. This match is done by means of a name lookup in the base language.

The list items that appear in a reduction clause may include array sections.

C++

If the type is a derived class, then any reduction identifier that matches its base classes is also a match, if no specific match for the type has been specified.

If the reduction identifier is not an *id-expression*, then it is implicitly converted to one by prepending the keyword operator (for example, **+** becomes *operator***+**).

If the reduction identifier is qualified then a qualified name lookup is used to find the declaration.

If the reduction identifier is unqualified then an *argument-dependent name lookup* must be performed using the type of each list item.

C++

If a list item is an array or array section, it will be treated as if a reduction clause would be applied to each separate element of the array section.

If a list item is an array section, the elements of any copy of the array section will be stored contiguously.

Fortran

If the original list item has the **POINTER** attribute, any copies of the list item are associated with private targets.

Fortran

Any copies of a list item associated with the reduction are initialized with the initializer value of the reduction identifier. Any copies are combined using the combiner associated with the reduction identifier.

Execution Model Events

The *reduction-begin* event occurs before a task begins to perform loads and stores that belong to the implementation of a reduction and the *reduction-end* event occurs after the task has completed loads and stores associated with the reduction. If a task participates in multiple reductions, each reduction may be bracketed by its own pair of *reduction-begin*/*reduction-end* events or multiple reductions may be bracketed by a single pair of events. The interval defined by a pair of *reduction-begin*/*reduction-end* events may not contain a task scheduling point.

Tool Callbacks

A thread dispatches a registered **ompt_callback_reduction** with **ompt_sync_region_reduction** in its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *reduction-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_reduction** with **ompt_sync_region_reduction** in its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of a *reduction-end* event in that thread. These callbacks occur in the context of the task that performs the reduction and has the type signature **ompt_callback_sync_region_t**.

Restrictions

Restrictions common to reduction clauses are as follows:

- Any number of reduction clauses can be specified on the directive, but a list item (or any array element in an array section) can appear only once in reduction clauses for that directive.
- For a reduction identifier declared in a **declare reduction** directive, the directive must appear before its use in a reduction clause.
- If a list item is an array section or an array element, its base expression must be a base language identifier.
- If a list item is an array section, it must specify contiguous storage and it cannot be a zero-length array section.
- If a list item is an array section or an array element, accesses to the elements of the array outside the specified array section or array element result in unspecified behavior.

C / C++

- The type of a list item that appears in a reduction clause must be valid for the reduction identifier. For a **max** or **min** reduction in C, the type of the list item must be an allowed arithmetic data type: **char**, **int**, **float**, **double**, or **_Bool**, possibly modified with **long**, **short**, **signed**, or **unsigned**. For a **max** or **min** reduction in C++, the type of the list item must be an allowed arithmetic data type: **char**, **wchar_t**, **int**, **float**, **double**, or **bool**, possibly modified with **long**, **short**, **signed**, or **unsigned**.
- A list item that appears in a reduction clause must not be **const**-qualified.
- The reduction identifier for any list item must be unambiguous and accessible.

C / C++

Fortran

- The type, type parameters and rank of a list item that appears in a reduction clause must be valid for the combiner expression and the initializer expression.
- A list item that appears in a reduction clause must be definable.
- A procedure pointer may not appear in a reduction clause.
- A pointer with the **INTENT(IN)** attribute may not appear in the reduction clause.
- An original list item with the **POINTER** attribute or any pointer component of an original list item that is referenced in a combiner expression must be associated at entry to the construct that contains the reduction clause. Additionally, the list item or the pointer component of the list item must not be deallocated, allocated, or pointer assigned within the region.

- An original list item with the **ALLOCATABLE** attribute or any allocatable component of an original list item that corresponds to a special variable identifier in the combiner expression or the initializer expression must be in the allocated state at entry to the construct that contains the reduction clause. Additionally, the list item or the allocatable component of the list item must be neither deallocated nor allocated, explicitly or implicitly, within the region.
- If the reduction identifier is defined in a **declare reduction** directive, the **declare reduction** directive must be in the same subprogram, or accessible by host or use association.
- If the reduction identifier is a user-defined operator, the same explicit interface for that operator must be accessible at the location of the **declare reduction** directive that defines the reduction identifier.
- If the reduction identifier is defined in a **declare reduction** directive, any procedure referenced in the **initializer** clause or the combiner expression must be an intrinsic function, or must have an explicit interface where the same explicit interface is accessible as at the **declare reduction** directive.

Fortran

Cross References

- **ompt_callback_sync_region_t**, see Section 19.5.2.13.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 19.4.4.11.
- **ompt_sync_region_reduction**, see Section 19.4.4.14.

5.5.7 Reduction Scoping Clauses

Reduction scoping clauses define the region in which a reduction is computed by tasks or SIMD lanes. All properties common to all reduction clauses, which are defined in Section 5.5.6, apply to reduction scoping clauses.

The number of copies created for each list item and the time at which those copies are initialized are determined by the particular reduction scoping clause that appears on the construct.

The time at which the original list item contains the result of the reduction is determined by the particular reduction scoping clause.

The location in the OpenMP program at which values are combined and the order in which values are combined are unspecified. Therefore, when comparing sequential and parallel executions, or when comparing one parallel execution to another (even if the number of threads used is the same), bitwise-identical results are not guaranteed to be obtained. Similarly, side effects (such as floating-point exceptions) may not be identical and may not take place at the same location in the OpenMP program.

To avoid data races, concurrent reads or updates of the original list item must be synchronized with the update of the original list item that occurs as a result of the reduction computation.

5.5.8 Reduction Participating Clauses

A reduction participating clause specifies a task or a SIMD lane as a participant in a reduction defined by a reduction scoping clause. All properties common to all reduction clauses, which are defined in Section 5.5.6, apply to reduction participating clauses.

Accesses to the original list item may be replaced by accesses to copies of the original list item created by a region that corresponds to a construct with a reduction scoping clause.

In any case, the final value of the reduction must be determined as if all tasks or SIMD lanes that participate in the reduction are executed sequentially in some arbitrary order.

5.5.9 `reduction` Clause

Name:	**Properties:**
`reduction`	data-environment attribute, data-sharing attribute, privatization, reduction scoping, reduction participating

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Modifiers:

Name	Modifies	Type	Properties
reduction-identifier	*list*	An OpenMP reduction identifier	required, ultimate
reduction-modifier	*list*	**Keyword: `default`**, **`inscan`**, **`task`**	default

Directives:

`do`, **`for`**, **`loop`**, **`parallel`**, **`scope`**, **`sections`**, **`simd`**, **`taskloop`**, **`teams`**

Semantics

The **`reduction`** clause is a reduction scoping clause and a reduction participating clause, as described in Section 5.5.7 and Section 5.5.8. For each list item, a private copy is created for each implicit task or SIMD lane and is initialized with the initializer value of the *reduction-identifier*. After the end of the region, the original list item is updated with the values of the private copies using the combiner associated with the *reduction-identifier*.

If *reduction-modifier* is not present or the **`default`** *reduction-modifier* is present, the behavior is as follows. For **`parallel`** and worksharing constructs, one or more private copies of each list item are created for each implicit task, as if the **`private`** clause had been used. For the **`simd`** construct, one or more private copies of each list item are created for each SIMD lane, as if the **`private`** clause had been used. For the **`taskloop`** construct, private copies are created according to the rules of the reduction scoping clauses. For the **`teams`** construct, one or more private copies of each list item are created for the initial task of each team in the league, as if the

private clause had been used. For the **loop** construct, private copies are created and used in the construct according to the description and restrictions in Section 5.3. At the end of a region that corresponds to a construct for which the **reduction** clause was specified, the original list item is updated by combining its original value with the final value of each of the private copies, using the combiner of the specified *reduction-identifier*.

If the **inscan** *reduction-modifier* is present, a scan computation is performed over updates to the list item performed in each logical iteration of the loop associated with the worksharing-loop, worksharing-loop SIMD, or **simd** construct (see Section 5.6). The list items are privatized in the construct according to the description and restrictions in Section 5.3. At the end of the region, each original list item is assigned the value described in Section 5.6.

If the **task** *reduction-modifier* is present for a **parallel** or worksharing construct, then each list item is privatized according to the description and restrictions in Section 5.3, and an unspecified number of additional private copies may be created to support task reductions. Any copies associated with the reduction are initialized before they are accessed by the tasks that participate in the reduction, which include all implicit tasks in the corresponding region and all participating explicit tasks that specify an **in_reduction** clause (see Section 5.5.11). After the end of the region, the original list item contains the result of the reduction.

If **nowait** is not specified for the construct, the reduction computation will be complete at the end of the region that corresponds to the construct; however, if the **reduction** clause is used on a construct to which **nowait** is also applied, accesses to the original list item will create a race and, thus, have unspecified effect unless synchronization ensures that they occur after all threads have executed all of their iterations or **section** constructs, and the reduction computation has completed and stored the computed value of that list item. This can be ensured simply through a barrier synchronization in most cases.

Restrictions

Restrictions to the **reduction** clause are as follows:

- All restrictions common to all reduction clauses, which are listed in Section 5.5.6, apply to this clause.
- A list item that appears in a **reduction** clause of a worksharing construct must be shared in the **parallel** region to which a corresponding worksharing region binds.
- If an array section or an array element appears as a list item in a **reduction** clause of a worksharing construct all threads that participate in the reduction must specify the same storage location.
- A list item that appears in a **reduction** clause with the **inscan** *reduction-modifier* must appear as a list item in an **inclusive** or **exclusive** clause on a **scan** directive enclosed by the construct.
- If the **inscan** *reduction-modifier* is specified, a **reduction** clause without the **inscan** *reduction-modifier* may not appear on the same construct.

- A **reduction** clause with the **task** *reduction-modifier* may only appear on a **parallel** construct, a worksharing construct or a combined or composite construct for which any of the aforementioned constructs is a constituent construct and **simd** or **loop** are not constituent constructs.

- A **reduction** clause with the **inscan** *reduction-modifier* may only appear on a worksharing-loop construct, a **simd** construct or a combined or composite construct for which any of the aforementioned constructs is a constituent construct and **distribute** is not a constituent construct.

- The **inscan** *reduction-modifier* cannot be specified on a construct for which the **ordered** or **schedule** clause is specified.

- A list item that appears in a **reduction** clause of the innermost enclosing worksharing or **parallel** construct may not be accessed in an explicit task generated by a construct for which an **in_reduction** clause over the same list item does not appear.

- The **task** *reduction-modifier* may not appear in a **reduction** clause if the **nowait** clause is specified on the same construct.

C / C++

- If a list item in a **reduction** clause on a worksharing construct has a reference type then it must bind to the same object for all threads of the team.

- If a list item in a **reduction** clause on a worksharing construct, is an array section or an array element then the base pointer must point to the same variable for all threads of the team.

- A variable of class type (or array thereof) that appears in a **reduction** clause with the **inscan** *reduction-modifier* requires an accessible, unambiguous default constructor for the class type. The number of calls to the default constructor while performing the scan computation is unspecified.

- A variable of class type (or array thereof) that appears in a **reduction** clause with the **inscan** *reduction-modifier* requires an accessible, unambiguous copy assignment operator for the class type. The number of calls to the copy assignment operator while performing the scan computation is unspecified.

C / C++

Cross References

- List Item Privatization, see Section 5.3.
- **ordered** clause, see Section 4.4.4.
- **scan** directive, see Section 5.6.
- **schedule** clause, see Section 11.5.3.
- **private** clause, see Section 5.4.3.

5.5.10 task_reduction Clause

Name:	Properties:
task_reduction	data-environment attribute, data-sharing attribute, privatization, reduction scoping

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Modifiers:

Name	Modifies	Type	Properties
reduction-identifier	*list*	An OpenMP reduction identifier	required, ultimate

Directives:

taskgroup

Semantics

The **task_reduction** clause is a reduction scoping clause, as described in 5.5.7, that specifies a reduction among tasks.

For each list item, the number of copies is unspecified. Any copies associated with the reduction are initialized before they are accessed by the tasks that participate in the reduction. After the end of the region, the original list item contains the result of the reduction.

Restrictions

Restrictions to the **task_reduction** clause are as follows:

- All restrictions common to all reduction clauses, which are listed in Section 5.5.6, apply to this clause.

5.5.11 in_reduction Clause

Name:	Properties:
in_reduction	data-environment attribute, data-sharing attribute, privatization, reduction participating

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Modifiers:

Name	Modifies	Type	Properties
reduction-identifier	*list*	An OpenMP reduction identifier	required, ultimate

Directives:
task, **taskloop**

Semantics

The **in_reduction** clause is a reduction participating clause, as described in Section 5.5.8, that specifies that a task participates in a reduction. For a given list item, the **in_reduction** clause defines a task to be a participant in a task reduction that is defined by an enclosing region for a matching list item that appears in a **task_reduction** clause or a **reduction** clause with **task** as the *reduction-modifier*, where either:

1. The matching list item has the same storage location as the list item in the **in_reduction** clause; or

2. A private copy, derived from the matching list item, that is used to perform the task reduction has the same storage location as the list item in the **in_reduction** clause.

For the **task** construct, the generated task becomes the participating task. For each list item, a private copy may be created as if the **private** clause had been used.

For the **target** construct, the target task becomes the participating task. For each list item, a private copy may be created in the data environment of the target task as if the **private** clause had been used. This private copy will be implicitly mapped into the device data environment of the target device, if the target device is not the parent device.

At the end of the task region, if a private copy was created its value is combined with a copy created by a reduction scoping clause or with the original list item.

Restrictions

Restrictions to the **in_reduction** clause are as follows:

- All restrictions common to all reduction clauses, which are listed in Section 5.5.6, apply to this clause.

- A list item that appears in a **task_reduction** clause or a **reduction** clause with **task** as the *reduction-modifier* that is specified on a construct that corresponds to a region in which the region of the participating task is closely nested must match each list item. The construct that corresponds to the innermost enclosing region that meets this condition must specify the same *reduction-identifier* for the matching list item as the **in_reduction** clause.

5.5.12 declare reduction Directive

Name: declare reduction	**Association:** none
Category: declarative	**Properties:** default

Arguments: `declare_reduction` (*reduction-identifier* : *typename-list* : *combiner*)

Name	Type	Properties
reduction-identifier	Identifier of type reduction	default
typename-list	List containing type-name list item	default
combiner	Expression of type combiner	default

Clauses:

`initializer`

Semantics

The **`declare reduction`** directive declares a *reduction-identifier* that can be used in a **`reduction`** clause as a user-defined reduction. The *reduction-identifier* and the type identify the **`declare reduction`** directive. The *reduction-identifier* can later be used in a **`reduction`** clause that uses variables of the types specified in the **`declare reduction`** directive. If the directive specifies several types then the behavior is as if a **`declare reduction`** directive was specified for each type. The visibility and accessibility of a user-defined reduction are the same as those of a variable declared at the same location in the program.

C++

The **`declare reduction`** directive can also appear at the locations in a program where a static data member could be declared. In this case, the visibility and accessibility of the declaration are the same as those of a static data member declared at the same location in the program.

C++

The enclosing context of the *combiner* and of the *initializer-expr* is that of the **`declare reduction`** directive. The *combiner* and the *initializer-expr* must be correct in the base language as if they were the body of a function defined at the same location in the program.

Fortran

If a type with deferred or assumed length type parameter is specified in a **`declare reduction`** directive, the *reduction-identifier* of that directive can be used in a reduction clause with any variable of the same type and the same kind parameter, regardless of the length type Fortran parameters with which the variable is declared.

If the *reduction-identifier* is the same as the name of a user-defined operator or an extended operator, or the same as a generic name that is one of the allowed intrinsic procedures, and if the operator or procedure name appears in an accessibility statement in the same module, the accessibility of the corresponding **`declare reduction`** directive is determined by the accessibility attribute of the statement.

If the *reduction-identifier* is the same as a generic name that is one of the allowed intrinsic procedures and is accessible, and if it has the same name as a derived type in the same module, the accessibility of the corresponding **`declare reduction`** directive is determined by the accessibility of the generic name according to the base language.

Fortran

Restrictions

Restrictions to the **declare reduction** directive are as follows:

- A *reduction-identifier* may not be re-declared in the current scope for the same type or for a type that is compatible according to the base language rules.
- The *typename-list* must not declare new types.

C / C++

- A type name in a **declare reduction** directive cannot be a function type, an array type, a reference type, or a type qualified with **const**, **volatile** or **restrict**.

C / C++

Fortran

- If the length type parameter is specified for a type, it must be a constant, a colon or an *****.
- If a type with deferred or assumed length parameter is specified in a **declare reduction** directive, no other **declare reduction** directive with the same type, the same kind parameters and the same *reduction-identifier* is allowed in the same scope.

Fortran

Cross References

- OpenMP combiner expressions, see Section 5.5.2.
- OpenMP initializer expressions, see Section 5.5.3.
- OpenMP reduction idenitifiers, see Section 5.5.1.
- **initializer** clause, see Section 5.5.5.

5.6 scan Directive

Name: scan **Category:** executable	**Association:** separating **Properties:** default

Separated Directives:

simd, **worksharing-loop**, **worksharing-loop SIMD**

Clauses:

exclusive, **inclusive**

Clause set:

Properties: fully exclusive, required	**Members:** inclusive, exclusive

Semantics

The **scan** directive separates the *final-loop-body* of an enclosing **simd** construct or worksharing-loop construct (or a composite construct that combines them) into a structured block sequence that serves as an *input phase* and a structured block sequence that serves as a *scan phase*. Thus, it specifies that a scan computation updates each list item on each logical iteration of the enclosing loop nest that is associated with the separated directive.

If the **inclusive** clause is specified, the input phase includes the preceding structured block sequence and that the scan phase includes the following structured block sequence and, thus, the directive specifies that an inclusive scan computation is performed for each list item of list. If the **exclusive** clause is specified, the input phase excludes the preceding structured block sequence and, instead includes the following structured block sequence, while the scan phase includes the preceding structured block sequence and, thus, the directive specifies that an exclusive scan computation is performed for each list item of *list*.

The input phase contains all computations that update the list item in the iteration, and the scan phase ensures that any statement that reads the list item uses the result of the scan computation for that iteration.

The list items that appear in an **inclusive** or **exclusive** clause may include array sections.

The result of a scan computation for a given iteration is calculated according to the last *generalized prefix sum* ($\mathtt{PRESUM}_{\mathtt{last}}$) applied over the sequence of values given by the original value of the list item prior to the loop and all preceding updates to the list item in the logical iteration space of the loop. The operation $\mathtt{PRESUM}_{\mathtt{last}}(op, a_1, \ldots, a_N)$ is defined for a given binary operator *op* and a sequence of *N* values $a_1, \ldots, a_N$ as follows:

- if $N = 1$, a_1
- if $N > 1$, $op(\ \mathtt{PRESUM}_{\mathtt{last}}(op, a_1, \ldots, a_K), \mathtt{PRESUM}_{\mathtt{last}}(op, a_L, \ldots, a_N)\ )$, where $1 \le K + 1 = L \le N$.

At the beginning of the input phase of each iteration, the list item is initialized with the value of the initializer expression of the *reduction-identifier* specified by the **reduction** clause on the separated construct. The *update value* of a list item is, for a given iteration, the value of the list item on completion of its input phase.

Let *orig-val* be the value of the original list item on entry to the separated construct. Let *combiner* be the combiner expression for the *reduction-identifier* specified by the **reduction** clause on the construct. Let u_I be the update value of a list item for iteration *I*. For list items that appear in an **inclusive** clause on the **scan** directive, at the beginning of the scan phase for iteration *I* the list item is assigned the result of the operation $\mathtt{PRESUM}_{\mathtt{last}}(\ combiner, orig\text{-}val, u_0, \ldots, u_I)$. For list items that appear in an **exclusive** clause on the **scan** directive, at the beginning of the scan phase for iteration $I = 0$ the list item is assigned the value *orig-val*, and at the beginning of the scan phase for iteration $I > 0$ the list item is assigned the result of the operation $\mathtt{PRESUM}_{\mathtt{last}}(\ combiner, orig\text{-}val, u_0, \ldots, u_{I-1})$.

For list items that appear in an **inclusive** clause, at the end of the separated construct, the original list item is assigned the private copy from the last logical iteration of the loops associated with the separated construct. For list items that appear in an **exclusive** clause, let *L* be the last logical iteration of the loops associated with the separated construct. At the end of the separated construct, the original list item is assigned the result of the operation $\texttt{PRESUM}_{\texttt{last}}$(*combiner*, *orig-val*, $u_0, \ldots, u_L$).

Restrictions

Restrictions to the **scan** directive are as follows:

- Exactly one **scan** directive must be associated with a directive on which a **reduction** clause with the **inscan** modifier is present.
- The loops that are associated with the directive to which the **scan** directive is associated must all be perfectly nested.
- Each list item that appears in the **inclusive** or **exclusive** clause must appear in a **reduction** clause with the **inscan** modifier on the separated construct.
- Each list item that appears in a **reduction** clause with the **inscan** modifier on the separated construct must appear in a clause on the separating **scan** directive.
- Cross-iteration dependences across different logical iterations must not exist, except for dependences for the list items specified in an **inclusive** or **exclusive** clause.
- Intra-iteration dependences from a statement in the structured block sequence that precede a **scan** directive to a statement in the structured block sequence that follows a **scan** directive must not exist, except for dependences for the list items specified in an **inclusive** or **exclusive** clause.
- The private copy of list items that appear in the **inclusive** or **exclusive** clause may not be modified in the *scan phase*.

Cross References

- Worksharing-loop construct, see Section 11.5.
- **exclusive** clause, see Section 5.6.2.
- **inclusive** clause, see Section 5.6.1.
- **reduction** clause, see Section 5.5.9.
- **simd** construct, see Section 10.4.

5.6.1 inclusive Clause

Name:	**Properties:**
inclusive	unique

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Directives:

scan

Semantics

The **inclusive** clause is used on a separating directive that separate a structured block into two structured block sequences. The clause determines the association of the structured block sequence that precedes the directive on which the clause appears to a phase of that directive. Specifically, the clause indicates that structure block sequence is included in the phase that is defined by the association for all list items in *list*.

Cross References

- **scan** directive, see Section 5.6.

5.6.2 exclusive Clause

Name:	**Properties:**
exclusive	unique

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Directives:

scan

Semantics

The **exclusive** clause is used on a separating directive that separate a structured block into two structured block sequences. The clause determines the association of the structured block sequence that precedes the directive on which the clause appears to a phase of that directive. Specifically, the clause indicates that structure block sequence is excluded from the phase that is defined by the association for all list items in *list*.

Cross References

- **scan** directive, see Section 5.6.

5.7 Data Copying Clauses

This section describes the **copyin** clause and the **copyprivate** clause. These two clauses support copying data values from private or threadprivate variables of an implicit task or thread to the corresponding variables of other implicit tasks or threads in the team.

Restrictions

Restrictions to the data copying clauses are as follows:

- All list items appearing in a clause must be visible, according to the scoping rules of the base language.
- A list item that specifies a given variable may not appear in more than one clause on the same directive.

5.7.1 `copyin` Clause

Name:	**Properties:**
`copyin`	data copying

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Directives:

`parallel`

Semantics

The **`copyin`** clause provides a mechanism to copy the value of a threadprivate variable of the primary thread to the threadprivate variable of each other member of the team that is executing the **`parallel`** region.

C / C++

The copy is performed after the team is formed and prior to the execution of the associated structured block. For variables of non-array type, the copy is by copy assignment. For an array of elements of non-array type, each element is copied as if by assignment from an element of the array of the primary thread to the corresponding element of the array of all other threads.

C / C++

C++

For class types, the copy assignment operator is invoked. The order in which copy assignment operators for different variables of the same class type are invoked is unspecified.

C++

Fortran

The copy is performed, as if by assignment, after the team is formed and prior to the execution of the associated structured block.

Named variables that appear in a threadprivate common block may be specified. The whole common block does not need to be specified.

On entry to any **parallel** region, each thread's copy of a variable that is affected by a **copyin** clause for the **parallel** region will acquire the type parameters, allocation, association, and definition status of the copy of the primary thread, according to the following rules:

- If the original list item has the **POINTER** attribute, each copy receives the same association status as that of the copy of the primary thread as if by pointer assignment.
- If the original list item does not have the **POINTER** attribute, each copy becomes defined with the value of the copy of the primary thread as if by intrinsic assignment unless the list item has a type bound procedure as a defined assignment. If the original list item that does not have the **POINTER** attribute has the allocation status of unallocated, each copy will have the same status.
- If the original list item is unallocated or unassociated, each copy inherits the declared type parameters and the default type parameter values from the original list item.

Fortran

Restrictions

Restrictions to the **copyin** clause are as follows:

- A list item that appears in a **copyin** clause must be threadprivate.

C / C++

- A variable of class type (or array thereof) that appears in a **copyin** clause requires an accessible, unambiguous copy assignment operator for the class type.

C / C++

Fortran

- A common block name that appears in a **copyin** clause must be declared to be a common block in the same scoping unit in which the **copyin** clause appears.
- A polymorphic variable with the **ALLOCATABLE** attribute cannot be a list item.

Fortran

Cross References

- **parallel** construct, see Section 10.1.
- **threadprivate** directive, see Section 5.2.

5.7.2 `copyprivate` Clause

Name: `copyprivate` **Properties:** end-clause, data copying

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Directives:

`single`

Semantics

The **`copyprivate`** clause provides a mechanism to use a private variable to broadcast a value from the data environment of one implicit task to the data environments of the other implicit tasks that belong to the parallel region. The effect of the **`copyprivate`** clause on the specified list items occurs after the execution of the structured block associated with the associated construct, and before any of the threads in the team have left the barrier at the end of the construct. To avoid data races, concurrent reads or updates of the list item must be synchronized with the update of the list item that occurs as a result of the **`copyprivate`** clause if, for example, the **`nowait`** clause is used to remove the barrier.

C / C++

In all other implicit tasks that belong to the parallel region, each specified list item becomes defined with the value of the corresponding list item in the implicit task associated with the thread that executed the structured block. For variables of non-array type, the definition occurs by copy assignment. For an array of elements of non-array type, each element is copied by copy assignment from an element of the array in the data environment of the implicit task that is associated with the thread that executed the structured block to the corresponding element of the array in the data environment of the other implicit tasks.

C / C++

C++

For class types, a copy assignment operator is invoked. The order in which copy assignment operators for different variables of class type are called is unspecified.

C++

Fortran

If a list item does not have the **POINTER** attribute, then in all other implicit tasks that belong to the parallel region, the list item becomes defined as if by intrinsic assignment with the value of the corresponding list item in the implicit task that is associated with the thread that executed the structured block. If the list item has a type bound procedure as a defined assignment, the assignment is performed by the defined assignment.

If the list item has the **POINTER** attribute, then, in all other implicit tasks that belong to the parallel region, the list item receives, as if by pointer assignment, the same association status of the corresponding list item in the implicit task that is associated with the thread that executed the structured block.

The order in which any final subroutines for different variables of a finalizable type are called is unspecified.

Fortran

Note – The **copyprivate** clause is an alternative to using a shared variable for the value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level).

Restrictions

Restrictions to the **copyprivate** clause are as follows:

- All list items that appear in the **copyprivate** clause must be either threadprivate or private in the enclosing context.
- A list item that appears in a **copyprivate** clause may not appear in a **private** or **firstprivate** clause on the associated construct.

C++

- A variable of class type (or array thereof) that appears in a **copyprivate** clause requires an accessible unambiguous copy assignment operator for the class type.

C++

Fortran

- A common block that appears in a **copyprivate** clause must be threadprivate.
- Pointers with the **INTENT(IN)** attribute may not appear in the **copyprivate** clause.
- Any list item with the **ALLOCATABLE** attribute must have the allocation status of allocated when the intrinsic assignment is performed.
- If a list item is a polymorphic variable with the **ALLOCATABLE** attribute, the behavior is unspecified.

Fortran

Cross References

- **parallel** construct, see Section 10.1.
- **single** construct, see Section 11.1.
- **threadprivate** directive, see Section 5.2.
- **private** clause, see Section 5.4.3.

5.8 Data-Mapping Attribute Rules, Clauses, and Directives

This section describes how the data-mapping and data-sharing attributes of any variable referenced in a **target** region are determined. When specified, explicit data-environment attribute clauses on **target** directives determine these attributes. Otherwise, the first matching rule from the following implicit data-mapping rules applies for variables referenced in a **target** construct that are not declared in the construct and do not appear as a list item or as a base variable or base pointer of a list item in one of the data-environment attribute clauses. References to structure elements or array elements are treated as references to the structure or array, respectively, for the purposes of determining implicit data-mapping or data-sharing attributes of variables in a **target** construct. A list item that appears in a **map** clause may also appear in a **use_device_ptr** clause or a **use_device_addr** clause.

- If a variable appears in an **enter** or **link** clause on a declare target directive that does not have a **device_type(nohost)** clause then it is treated as if it had appeared in a **map** clause with a *map-type* of **tofrom**.
- If a variable is the base variable of a list item in a **reduction**, **lastprivate** or **linear** clause on a combined target construct then the list item is treated as if it had appeared in a **map** clause with a *map-type* of **tofrom** if Section 17.2 specifies this behavior.
- If a variable is the base variable of a list item in an **in_reduction** clause on a **target** construct then it is treated as if the list item had appeared in a **map** clause with a *map-type* of **tofrom** and a *map-type-modifier* of **always**.
- If a **defaultmap** clause is present for the category of the variable and specifies an implicit behavior other than **default**, the data-mapping attribute is determined by that clause.

C++

- If the **target** construct is within a class non-static member function, and a variable is an accessible data member of the object for which the non-static data member function is invoked, the variable is treated as if the **this[:1]** expression had appeared in a **map** clause with a *map-type* of **tofrom**. Additionally, if the variable is of type pointer or reference to pointer, it is also treated as if it had appeared in a **map** clause as a zero-length array section.

- If the **this** keyword is referenced inside a **target** construct within a class non-static member function, it is treated as if the **this[:1]** expression had appeared in a **map** clause with a *map-type* of **tofrom**.

C++

C / C++

- A variable that is of type pointer, but not a function pointer or (for C++) a pointer to a member function, is treated as if it is the base pointer of a zero-length array section that had appeared as a list item in a **map** clause.

C / C++

C++

- A variable that is of type reference to pointer, but not a function pointer or a reference to a pointer to a member function is treated as if it had appeared in a **map** clause as a zero-length array section.

C++

- If a variable is not a scalar then it is treated as if it had appeared in a **map** clause with a *map-type* of **tofrom**.

Fortran

- If a scalar variable has the **TARGET**, **ALLOCATABLE** or **POINTER** attribute then it is treated as if it had appeared in a **map** clause with a *map-type* of **tofrom**.

Fortran

- If none of the above rules applies then a scalar variable is not mapped, but instead has an implicit data-sharing attribute of firstprivate (see Section 5.1.1).

5.8.1 OpenMP Mapper Identifiers and mapper Modifiers

OpenMP mapper identifiers can be used to uniquely identify the mapper used in a **map** or data-motion clause through a mapper modifier, which is a unique, complex modifier. A **declare mapper** directive defines a mapper identifier that can later be specified in a mapper modifier as its *modifier-parameter-specification*. Each OpenMP mapper identifier is a base-language identifier or **default** where **default** is the default mapper for all types.

A non-structure type *T* has a predefined default mapper that is defined as if by the following **declare mapper** directive:

C / C++

```
#pragma omp declare mapper(T v) map(tofrom: v)
```

C / C++

Fortran

```
!$omp declare mapper(T :: v) map(tofrom: v)
```

Fortran

A structure type *T* has a predefined default mapper that is defined as if by a **declare mapper** directive that specifies *v* in a **map** clause with the **alloc** *map-type* and each structure element of *v* in a **map** clause with the **tofrom** *map-type*.

A **declare mapper** directive that uses the **default** *mapper-identifier* overrides the predefined default mapper for the given type, making it the default mapper for variables of that type.

5.8.2 map Clause

Name:	**Properties:**
map	data-environment attribute, data-mapping attribute

Arguments:

Name	Type	Properties
locator-list	List containing locator list item	default

Modifiers:

Name	Modifies	Type	Properties
map-type-modifier	*locator-list*	**Keyword: always**, **close**, **present**	default
mapper-modifier	*locator-list*	mapper modifier	default
iterator-modifier	*locator-list*	iterator modifier	default
map-type	*locator-list*	**Keyword: alloc**, **delete**, **from**, **release**, **to**, **tofrom**	ultimate

Directives:

declare mapper, **target**, **target data**, **target enter data**, **target exit data**

Semantics

The **map** clause specifies how an original list item is mapped from the current task's data environment to a corresponding list item in the device data environment of the device identified by the construct. If a *map-type* is not specified, the *map-type* defaults to **tofrom**. The **map** clause is *map-entering* if the *map-type* is **to**, **tofrom** or **alloc**. The **map** clause is *map-exiting* if the *map-type* is **from**, **tofrom**, **release** or **delete**.

The list items that appear in a **map** clause may include array sections and structure elements. A list item in a **map** clause may reference iterators defined by the *iterators-modifier*. A list item may appear more than once in the **map** clauses that are specified on the same directive.

If a mapper modifier is not present, the behavior is as if a mapper modifier was specified with the **default** parameter. The map behavior of a list item in a **map** clause is modified by a visible user-defined mapper (see Section 5.8.10) if the mapper has the same *mapper-identifier* as the *mapper-identifier* in the mapper modifier and is specified for a type that matches the type of the list item. Otherwise, the predefined default mapper for the type of the list item applies. The effect of the mapper is to remove the list item from the **map** clause, if the **present** modifier does not also appear, and to apply the clauses specified in the declared mapper to the construct on which the **map** clause appears. In the clauses applied by the mapper, references to *var* are replaced with references to the list item and the *map-type* is replaced with a final map type that is determined according to the rules of map-type decay (see Section 5.8.10).

A list item that is an array or array section of a type for which a user-defined mapper exists is mapped as if the map type decays to **alloc**, **release**, or **delete**, and then each array element is mapped with the original map type, as if by a separate construct, according to the mapper.

Fortran

If a component of a derived type list item is a **map** clause list item that results from the predefined default mapper for that derived type, and if the derived type component is not an explicit list item or the base expression of an explicit list item in a **map** clause on the construct, then:

- If it has the **POINTER** attribute, the **map** clause treats its association status as if it is undefined; and
- If it has the **ALLOCATABLE** attribute and an allocated allocation status, and it is present in the device data environment when the construct is encountered, the **map** clause may treat its allocation status as if it is unallocated if the corresponding component does not have allocated storage.

If a list item in a **map** clause is an associated pointer and the pointer is not the base pointer of another list item in a **map** clause on the same construct, then it is treated as if its pointer target is implicitly mapped in the same clause. For the purposes of the **map** clause, the mapped pointer target is treated as if its base pointer is the associated pointer.

Fortran

For **map** clauses on map-entering constructs, if any list item has a base pointer for which a corresponding pointer exists in the data environment upon entry to the region and either a new list item or the corresponding pointer is created in the device data environment on entry to the region, then:

C / C++

1. The corresponding pointer variable is assigned an address such that the corresponding list item can be accessed through the pointer in a **target** region.

C / C++

Fortran

1. The corresponding pointer variable is associated with a pointer target that has the same rank and bounds as the pointer target of the original pointer, such that the corresponding list item can be accessed through the pointer in a **target** region.

Fortran

2. The corresponding pointer variable becomes an attached pointer for the corresponding list item.

3. If the original base pointer and the corresponding attached pointer share storage, then the original list item and the corresponding list item must share storage.

C++

If a *lambda* is mapped explicitly or implicitly, variables that are captured by the *lambda* behave as follows:

- The variables that are of pointer type are treated as if they had appeared in a **map** clause as zero-length array sections; and
- The variables that are of reference type are treated as if they had appeared in a **map** clause.

If a member variable is captured by a *lambda* in class scope, and the *lambda* is later mapped explicitly or implicitly with its full static type, the **this** pointer is treated as if it had appeared on a **map** clause.

C++

If a **map** clause with a **present** *map-type-modifier* appears on a construct then on entry to the region if the corresponding list item does not appear in the device data environment then the behavior is as if an **error** directive for which *sev-level* is **fatal** and *action-time* is **execution** is encountered.

The **map** clauses on a construct collectively determine the set of *mappable storage blocks* for that construct. All **map** clause list items that have the same containing structure or share storage result in a single mappable storage block that encompasses the storage of the list items. The storage for each other **map** clause list item becomes a distinct mappable storage block.

For each mappable storage block that is determined by the **map** clauses on a map-entering construct, on entry to the region the following sequence of steps occurs as if they are performed as a single atomic operation:

1. If a corresponding storage block is not present in the device data environment then:

   a) A corresponding storage block, which may be the same as the original storage block, is created in the device data environment of the device;

   b) The corresponding storage block receives a reference count that is initialized to zero. This reference count also applies to any part of the corresponding storage block.

2. The reference count of the corresponding storage block is incremented by one.

3. For each **map** clause list item on the construct that is encompassed by the mappable storage block:

   a) If the reference count of the corresponding storage block is one, a new list item with language-specific attributes derived from the original list item is created in the corresponding storage block. The reference count of the new list item is always equal to the reference count of its storage.

b) If the reference count of the corresponding list item is one or if the **always** *map-type-modifier* is specified, and if the *map-type* is **to** or **tofrom**, the corresponding list item is updated as if the list item appeared in a **to** clause on a **target update** directive.

Note – If the effect of the **map** clauses on a construct would assign the value of an original list item to a corresponding list item more than once, then an implementation is allowed to ignore additional assignments of the same value to the corresponding list item.

In all cases on entry to the region, concurrent reads or updates of any part of the corresponding list item must be synchronized with any update of the corresponding list item that occurs as a result of the **map** clause to avoid data races.

The original and corresponding list items may share storage such that writes to either item by one task followed by a read or write of the other item by another task without intervening synchronization can result in data races. They are guaranteed to share storage if the **map** clause appears on a **target** construct that corresponds to an inactive **target** region, or if it appears on a mapping-only construct that applies to the device data environment of the host device.

If corresponding storage for a mappable storage block derived from **map** clauses on a map-exiting csontruct is not present in the device data environment on exit from the region, the mappable storage block is ignored. For each mappable storage block that is determined by the **map** clauses on a map-exiting construct, on exit from the region the following sequence of steps occurs as if performed as a single atomic operation:

1. For each **map** clause list item that is encompassed by the mappable storage block:

   a) If the reference count of the corresponding list item is one or if the **always** *map-type-modifier* is specified, and if the *map-type* is **from** or **tofrom**, the original list item is updated as if the list item appeared in a **from** clause on a **target update** directive.

2. If the *map-type* is not **delete** and the reference count of the corresponding storage block is finite then the reference count is decremented by one.

3. If the *map-type* is **delete** and the reference count of the corresponding storage block is finite then the reference count is set to zero.

4. If the reference count of the corresponding storage block is zero, all storage to which that reference count applies is removed from the device data environment.

Note – If the effect of the **map** clauses on a construct would assign the value of a corresponding list item to an original list item more than once, then an implementation is allowed to ignore additional assignments of the same value to the original list item.

In all cases on exit from the region, concurrent reads or updates of any part of the original list item must be synchronized with any update of the original list item that occurs as a result of the **map** clause to avoid data races.

If a single contiguous part of the original storage of a list item with an implicit data-mapping attribute has corresponding storage in the device data environment prior to a task encountering the construct that is associated with the **map** clause, only that part of the original storage will have corresponding storage in the device data environment as a result of the **map** clause.

If a list item with an implicit data-mapping attribute does not have any corresponding storage in the device data environment prior to a task encountering the construct associated with the **map** clause, and one or more contiguous parts of the original storage are either list items or base pointers to list items that are explicitly mapped on the construct, only those parts of the original storage will have corresponding storage in the device data environment as a result of the **map** clauses on the construct.

C / C++

If a new list item is created then the new list item will have the same static type as the original list item, and language-specific attributes of the new list item, including size and alignment, are determined by that type.

C / C++

C++

If corresponding storage that differs from the original mappable storage block is created in a device data environment, all new list items that are created in that corresponding storage are default initialized. Default initialization for new list items of class type, including their data members, is performed as if with an implicity-declared default constructor and as if non-static data member initializers are ignored.

If the type of a new list item is a reference to a type *T* then it is initialized to refer to the object in the device data environment that corresponds to the object referenced by the original list item. The effect is as if the object were mapped through a pointer with an array section of length one and elements of type *T*.

C++

Fortran

If a new list item is created then the new list item will have the same type, type parameter, and rank as the original list item. The new list item inherits all default values for the type parameters from the original list item.

If the allocation status of an original list item that has the **ALLOCATABLE** attribute is changed while a corresponding list item is present in the device data environment, the allocation status of the corresponding list item is unspecified until the list item is again mapped with an **always** modifier on entry to a map-entering region.

Fortran

The **close** *map-type-modifier* is a hint to the runtime to allocate memory close to the target device.

Execution Model Events

The *target-map* event occurs in a thread that executes the outermost region that corresponds to an encountered device construct with a **map** clause, after the *target-task-begin* event for the device construct and before any mapping operations are performed.

The *target-data-op-begin* event occurs before a thread initiates a data operation on the target device that is associated with a **map** clause, in the outermost region that corresponds to the encountered construct.

The *target-data-op-end* event occurs after a thread initiates a data operation on the target device that is associated with a **map** clause, in the outermost region that corresponds to the encountered construct.

Tool Callbacks

A thread dispatches one or more registered **ompt_callback_target_map** or **ompt_callback_target_map_emi** callbacks for each occurrence of a *target-map* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_map_t** or **ompt_callback_target_map_emi_t**, respectively.

A thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_begin** as its endpoint argument for each occurrence of a *target-data-op-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_end** as its endpoint argument for each occurrence of a *target-data-op-end* event in that thread. These callbacks have type signature **ompt_callback_target_data_op_emi_t**.

A thread dispatches a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-op-end* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**.

Restrictions

Restrictions to the **map** clause are as follows:

- Two list items of the **map** clauses on the same construct must not share original storage unless they are the same list item or unless one is the containing structure of the other.
- If the same list item appears more than once in **map** clauses on the same construct, the **map** clauses must specify the same **mapper** modifier.
- If a list item is an array section, it must specify contiguous storage.
- If an expression that is used to form a list item in a **map** clause contains an iterator identifier, the list item instances that would result from different values of the iterator must not have the same containing array and must not have base pointers that share original storage.

- If multiple list items are explicitly mapped on the same construct and have the same containing array or have base pointers that share original storage, and if any of the list items do not have corresponding list items that are present in the device data environment prior to a task encountering the construct, then the list items must refer to the same array elements of either the containing array or the implicit array of the base pointers.
- If any part of the original storage of a list item with an explicit data-mapping attribute has corresponding storage in the device data environment prior to a task encountering the construct associated with the **map** clause, all of the original storage must have corresponding storage in the device data environment prior to the task encountering the construct.
- If an array appears as a list item in a **map** clause, multiple parts of the array have corresponding storage in the device data environment prior to a task encountering the construct associated with the **map** clause, and the corresponding storage for those parts was created by maps from more than one earlier construct, the behavior is unspecified.
- If a list item is an element of a structure, and a different element of the structure has a corresponding list item in the device data environment prior to a task encountering the construct associated with the **map** clause, then the list item must also have a corresponding list item in the device data environment prior to the task encountering the construct.
- A list item must have a mappable type.
- Threadprivate variables cannot appear in a **map** clause.
- If a mapper modifier appears in a **map** clause, the type on which the specified mapper operates must match the type of the list items in the clause.
- Memory spaces and memory allocators cannot appear as a list item in a **map** clause.

C++

- If a list item has a polymorphic class type and its static type does not match its dynamic type, the behavior is unspecified if the **map** clause is specified on a map-entering construct and a corresponding list item is not present in the device data environment prior to a task encountering the construct.
- No type mapped through a reference can contain a reference to its own type, or any references to types that could produce a cycle of references.
- If a list item is a *lambda*, any pointers and references captured by the *lambda* must have the corresponding list item in the device data environment prior to the task encountering the construct.

C++

C / C++

- A list item cannot be a variable that is a member of a structure of a union type.
- A bit-field cannot appear in a **map** clause.
- A pointer that has a corresponding attached pointer must not be modified for the duration of the lifetime of the list item to which the corresponding pointer is attached in the device data environment.

C / C++

Fortran

- If a list item of a **map** clause is an allocatable variable or is the subobject of an allocatable variable, the original allocatable variable may not be allocated, deallocated or reshaped while the corresponding allocatable variable has allocated storage.
- A pointer that has a corresponding attached pointer and is associated with a given pointer target must not become associated with a different pointer target for the duration of the lifetime of the list item to which the corresponding pointer is attached in the device data environment.
- If an array section is mapped and the size of the section is smaller than that of the whole array, the behavior of referencing the whole array in the **target** region is unspecified.
- A list item must not be a whole array of an assumed-size array.
- A list item must not be a complex part designator.

Fortran

Cross References

- **ompt_callback_target_data_op_t** or **ompt_callback_target_data_op_emi_t** callback type, see Section 19.5.2.25.
- **ompt_callback_target_map_t** or **ompt_callback_target_map_emi_t** callback type, see Section 19.5.2.27.
- Array sections, see Section 3.2.4.
- Iterators, see Section 3.2.5.
- **declare mapper** directive, see Section 5.8.10.

5.8.3 `is_device_ptr` Clause

Name:	**Properties:**
`is_device_ptr`	default

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Directives:
`dispatch`, **`target`**

Semantics

The **`is_device_ptr`** clause indicates that its list items are device pointers. Support for device pointers created outside of OpenMP, specifically outside of any OpenMP mechanism that returns a device pointer, is implementation defined.

If the **`is_device_ptr`** clause is specified on a **`target`** construct, each list item privatized inside the construct and the new list item is initialized to the device address to which the original list item refers.

Fortran

If the **`is_device_ptr`** clause is specified on a **`target`** construct, if any list item is not of type **`C_PTR`**, the behavior is as if the list item appeared in a **`has_device_addr`** clause. Support for such list items in an **`is_device_ptr`** clause is deprecated.

Fortran

Restrictions

Restrictions to the **`is_device_ptr`** clause are as follows:

- Each list item must be a valid device pointer for the device data environment.

C

- Each list item must have a type of pointer or array.

C

C++

- Each list item must have a type of pointer, array, reference to pointer or reference to array.

C++

Fortran

- Each list item must be of type **`C_PTR`** unless the clause appears on a **`target`** directive; the use of list items on the **`target`** directive that are not of type **`C_PTR`** has been deprecated.

Fortran

Cross References

- **dispatch** construct, see Section 7.6.
- **target** construct, see Section 13.8.

5.8.4 use_device_ptr Clause

Name:	Properties:
use_device_ptr	default

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Directives:

target data

Semantics

C / C++

If a list item that appears in a **use_device_ptr** clause is a pointer to an object that is mapped to the device data environment, references to the list item in the structured block that is associated with the construct on which the clause appears are converted into references to a device pointer that is local to the structured block and that refers to the device address of the corresponding object. If the list item does not point to a mapped object, it must contain a valid device address for the target device, and the list item references are instead converted to references to a local device pointer that refers to this device address.

C / C++

Fortran

If a list item that appears in a **use_device_ptr** clause is of type **C_PTR** and points to a data entity that is mapped to the device data environment, references to the list item in the structured block that is associated with the construct on which the clause appears are converted into references to a device pointer that is local to the structured block and that refers to the device address of the corresponding entity. If a list item of type **C_PTR** does not point to a mapped object, it must contain a valid device address for the target device, and the list item references are instead converted to references to a local device pointer that refers to this device address. If a list item in a **use_device_ptr** clause is not of type **C_PTR**, the behavior is as if the list item appeared in a **use_device_addr** clause. Support for such list items in a **use_device_ptr** clause is deprecated.

Fortran

Restrictions

Restrictions to the **use_device_ptr** clause are as follows:

- Each list item must not be a structure element.

C / C++

- Each list item must be a pointer for which the value is the address of an object that has corresponding storage in the device data environment or is accessible on the target device.

C / C++

Fortran

- The value of a list item that is of type **C_PTR** must be the address of a data entity that has corresponding storage in the device data environment or is accessible on the target device.

Fortran

Cross References

- **target data** construct, see Section 13.5.

5.8.5 has_device_addr Clause

Name:	**Properties:**
has_device_addr	default

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Directives:

target

Semantics

The **has_device_addr** clause indicates that its list items already have device addresses and therefore they may be directly accessed from a target device. If the device address of a list item is not for the device on which the region that is associated withe construct on which the clause appears executes, accessing the list item inside the region results in unspecified behavior. The list items may include array sections.

Restrictions

Restrictions to the **has_device_addr** clause are as follows:

- Each list item must have a valid device address for the device data environment.

Cross References

- **target** construct, see Section 13.8.

5.8.6 `use_device_addr` Clause

Name: `use_device_addr` **Properties:** default

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Directives:

`target data`

Semantics

If a list item has corresponding storage in the device data environment, references to the list item in the structured block that is associated with the construct on which the **`use_device_addr`** clause appears are converted into references to the corresponding list item. If the list item is not a mapped list item, it is assumed to be accessible on the target device. Inside the structured block, the list item has a device address and its storage may not be accessible from the host device. The list items that appear in a **`use_device_addr`** clause may include array sections.

C / C++

If a list item in a **`use_device_addr`** clause is an array section that has a base pointer, the effect of the clause is to convert the base pointer to a pointer that is local to the structured block and that contains the device address. This conversion may be elided if the list item was not already mapped.

C / C++

Restrictions

Restrictions to the **`use_device_addr`** clause are as follows:

- Each list item must have a corresponding list item in the device data environment or be accessible on the target device.
- Each list item must not be a structure element.

C / C++

- If a list item is an array section, the base expression must be a base language identifier.

C / C++

Fortran

- If a list item is an array section, the designator of the base expression must be a name without any selectors.

Fortran

Cross References

- **`target data`** construct, see Section 13.5.

5.8.7 `link` Clause

Name: **`link`**

Properties: default

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Directives:

`begin declare target`, **`declare target`**

Semantics

Including list items in a **`link`** clause supports compilation of functions called in a **`target`** region that refer to the list items. The **`declare target`** directive on which the clause appears does not map the list items. Instead, they are mapped according to the data mapping rules described in Section 5.8.

Cross References

- **`declare target`** directive, see Section 7.8.1.

C / C++

5.8.8 Pointer Initialization for Device Data Environments

This section describes how a pointer that is predetermined firstprivate for a **`target`** construct may be assigned an initial value that is the address of an object that exists in a device data environment and corresponds to a *matching mapped list item*.

All previously mapped list items that have corresponding storage in a given device data environment constitute the set of currently mapped list items. If a currently mapped list item has a base pointer, the *base address* of the currently mapped list item is the value of its base pointer. Otherwise, the base address is determined by the following steps:

1. Let *X* refer to the currently mapped list item.
2. If *X* refers to an array section or array element, let *X* refer to its base array.
3. If *X* refers to a structure element, let *X* refer to its containing structure and return to step 2.
4. The base address for the currently mapped list item is the address of *X*.

Additionally, each currently mapped list item has a *starting address* and an *ending address*. The starting address is the address of the first storage location associated with the list item, and the *ending address* is the address of the storage location that immediately follows the last storage location associated with the list item.

The *mapped address range* of the currently mapped list item is the range of addresses that starts from the starting address and ends with the ending address. The *extended address range* of the currently mapped list item is the range of addresses that starts from the minimum of the starting address and the base address and that ends with the maximum of the ending address and the base address.

If the value of a given pointer is in the mapped address range of a currently mapped list item then that currently mapped list item is a matching mapped list item. Otherwise, if the value of the pointer is in the extended address range of a currently mapped list item then that currently mapped list item is a matching mapped list item.

If multiple matching mapped list items are found and they all appear as part of the same containing structure, the one that has the lowest starting address is treated as the sole matching mapped list item. Otherwise, if multiple matching mapped list items are found then the behavior is unspecified.

If a matching mapped list item is found, the initial value that is assigned to the pointer is a device address such that the corresponding list item in the device data environment can be accessed through the pointer in a **`target`** region.

If a matching mapped list item is not found, the pointer retains its original value as per the firstprivate semantics described in Section 5.4.4.

Cross References

- **`map`** clause, see Section 5.8.2.
- **`requires`** directive, see Section 8.2.
- **`target`** construct, see Section 13.8.

C / C++

5.8.9 `defaultmap` Clause

Name:	**Properties:**
`defaultmap`	unique

Arguments:

Name	Type	Properties
variable-category	**Keyword:** **`aggregate`**, **`all`**, **`allocatable`**, **`pointer`**, **`scalar`**	optional

Modifiers:

Name	Modifies	Type	Properties
implicit-behavior	*variable-category*	**Keyword:** **`alloc`**, **`default`**, **`firstprivate`**, **`from`**, **`none`**, **`present`**, **`to`**, **`tofrom`**	required, ultimate

Directives:

`target`

Semantics

The **defaultmap** clause explicitly determines the data-mapping attributes of variables that are referenced in a **target** construct for which the data-mapping attributes would otherwise be implicitly determined (see Section 5.8). If no *variable-category* is specified in the clause then the effect is as if **all** was specified for the *variable-category*.

The effect of the **defaultmap** clause is as follows:

- If *variable-category* is **all**, all variables that are referenced in the construct have the data-mapping or data-sharing attribute specified by *implicit-behavior*.
- If *variable-category* is **scalar**, all scalar variables of non-pointer type or all non-pointer non-allocatable scalar variables that have an implicitly determined data-mapping or data-sharing attribute have the data-mapping or data-sharing attribute specified by *implicit-behavior*.
- If *variable-category* is **aggregate** or **allocatable**, all aggregate or allocatable variables that have an implicitly determined data-mapping or data-sharing attribute have the data-mapping or data-sharing attribute specified by *implicit-behavior*.
- If *variable-category* is **pointer**, all variables of pointer type or with the **POINTER** attribute that have implicitly determined data-mapping or data-sharing attributes have the data-mapping or data-sharing attribute specified by *implicit-behavior*.

If *implicit-behavior* is **none**, each variable referenced in the construct that does not have a predetermined data-sharing attribute and does not appear in an **enter** or **link** clause on a declare target directive must be listed in a data-mapping attribute clause, a data-sharing attribute clause (including a data-sharing attribute clause on a combined construct where **target** is one of the constituent constructs), an **is_device_ptr** clause or a **has_device_addr** clause. If *implicit-behavior* is **default**, then the clause has no effect for the variables in the category specified by *variable-category*. If *implicit-behavior* is **present**, each variable referenced in the construct in the category specified by *variable-category* is treated as if it had been listed in a **map** clause with the *map-type* of **alloc** and *map-type-modifier* of **present**.

Restrictions

Restrictions to the **defaultmap** clause are as follows:

C / C++

- The specified *variable-category* must not be **allocatable**.

C / C++

Cross References

- **target** construct, see Section 13.8.

5.8.10 `declare mapper` Directive

Name: `declare mapper` **Category:** declarative	**Association:** none **Properties:** default

Arguments: `declare_mapper(`*[mapper-identifier* `:` *] type var*`)`

Name	Type	Properties
mapper-identifier	Identifier of type mapper	default
type	type-name	default
var	Identifier of type base language	default

Clauses:

`map`

Semantics

User-defined mappers can be defined using the **`declare mapper`** directive. The *type* and an optional *mapper-identifier* uniquely identify the mapper for use in a **`map`** clause or motion clause later in the program. The visibility and accessibility of this declaration are the same as those of a variable declared at the same location in the program.

If *mapper-identifier* is not specified, the behavior is as if *mapper-identifier* is **`default`**.

The variable declared by *var* is available for use in all **`map`** clauses on the directive, and no part of the variable to be mapped is mapped by default.

The effect that a user-defined mapper has on either a **`map`** clause that maps a list item of the given base language type or a motion clause that invokes the mapper and updates a list item of the given base language type is to replace the map or update with a set of **`map`** clauses or updates derived from the **`map`** clauses specified by the mapper, as described in Section 5.8.2 and Section 13.9.

The final map types that a mapper applies for a **`map`** clause that maps a list item of the given type are determined according to the rules of map-type decay, defined according to Table 5.3. Table 5.3 shows the final map type that is determined by the combination of two map types, where the rows represent the map type specified by the mapper and the columns represent the map type specified by a **`map`** clause that invokes the mapper. For a **`target exit data`** construct that invokes a mapper with a **`map`** clause that has the **`from`** map type, if a **`map`** clause in the mapper specifies an **`alloc`** or **`to`** map type then the result is a **`release`** map type.

A list item in a **`map`** clause that appears on a **`declare mapper`** directive may include array sections.

All **`map`** clauses that are introduced by a mapper are further subject to mappers that are in scope, except a **`map`** clause with list item *var* maps *var* without invoking a mapper.

TABLE 5.3: Map-Type Decay of Map Type Combinations

	alloc	to	from	tofrom	release	delete
alloc	alloc	alloc	alloc (release)	alloc	release	delete
to	alloc	to	alloc (release)	to	release	delete
from	alloc	alloc	from	from	release	delete
tofrom	alloc	to	from	tofrom	release	delete

C++

The **declare mapper** directive can also appear at locations in the program at which a static data member could be declared. In this case, the visibility and accessibility of the declaration are the same as those of a static data member declared at the same location in the program.

C++

Restrictions

Restrictions to the **declare mapper** directive are as follows:

- No instance of *type* can be mapped as part of the mapper, either directly or indirectly through another base language type, except the instance *var* that is passed as the list item. If a set of **declare mapper** directives results in a cyclic definition then the behavior is unspecified.
- The *type* must not declare a new base language type.
- At least one **map** clause that maps *var* or at least one element of *var* is required.
- List items in **map** clauses on the **declare mapper** directive may only refer to the declared variable *var* and entities that could be referenced by a procedure defined at the same location.
- For each **map** clause, each *map-type-modifier* can appear at most once.
- Neither the **release** or **delete** *map-type* may be specified on any **map** clause.
- If a mapper modifier is specified for a **map** clause, its parameter must be **default**.
- Multiple **declare mapper** directives that specify the same *mapper-identifier* for the same base language type or for compatible base language types, according to the base language rules, may not appear in the same scope.

C

- *type* must be a **struct** or **union** type.

C

C++

- *type* must be a **struct**, **union**, or **class** type.

C++

Fortran

- *type* must not be an intrinsic type or an abstract type.

Fortran

Cross References

- **map** clause, see Section 5.8.2.
- **target update** construct, see Section 13.9.

5.9 Data-Motion Clauses

Data-motion clauses specify data movement between a device set that is specified by the construct on which they appear. One member of that device set is always the *encountering device*, which is the device on which the encountering task for that construct executes. How the other device(s), which are the *targeted device(s)* are determined is defined with the construct but is generally specified through a **device** clause. The *clause-name* of a data-motion clause specifies the direction of the data movement relative to the targeted device(s).

A data-motion clause specifies an OpenMP locator list as its argument. A corresponding list item and an original list item exist for each list item. If the corresponding list item is not present in the device data environment and the **present** modifier is not specified in the clause then no assignment occurs between the corresponding and original list items. Otherwise, each corresponding list item in the device data environment has an original list item in the data environment of the encountering task. Assignment is performed to either the orginal or corresponding list item as specified with the specific data-motion clauses. List items may reference iterators defined by *item-modifier*. The list items may include array sections with *stride* expressions.

C / C++

The list items may use shape-operators.

C / C++

If a list item is an array or array section then it is treated as if it is replaced by each of its array elements in the clause.

If *mapper-modifier* is not specified, the behavior is as if *mapper-identifier* is **default**. The effect of a data-motion clause on a list item is modified by a visible user-defined mapper if *mapper-identifier* is specified for a type that matches the type of the list item. Otherwise, the predefined default mapper for the type of the list item applies. Each list item is replaced with the list items that the given mapper specifies are to be mapped with a map type that is compatible with the data movement direction associated with the clause.

If a **present** *expectation* is specified and the corresponding list item is not present in the device data environment then the behavior is as if an **error** directive for which *sev-level* is **fatal** and *action-time* is **execution** is encountered. For a list item that is replaced with a set of list items as a result of a user-defined mapper, the *expectation* only applies to those mapper list items that share storage with the original list item.

Fortran

If a list item or a subobject of a list item has the **ALLOCATABLE** attribute, its assignment is performed only if its allocation status is allocated and only with respect to the allocated storage. If a list item has the **POINTER** attribute and its association status is associated, the effect is as if the assignment is performed with respect to the pointer target.

On exit from the associated region, if the corresponding list item is an attached pointer, the original list item, if associated, will be associated with the same pointer target with which it was associated on entry to the region and the corresponding list item, if associated, will be associated with the same pointer target with which it was associated on entry to the region.

Fortran

C / C++

On exit from the associated region, if the corresponding list item is an attached pointer, the original list item will have the value it had on entry to the region and the corresponding list item will have the value it had on entry to the region.

C / C++

For each list item that is not an attached pointer, the value of the assigned list item is assigned the value of the other list item. To avoid data races, concurrent reads or updates of the assigned list item must be synchronized with the update of an assigned list item that occurs as a result of a data-motion clause.

Restrictions

Restrictions to data-motion clauses are as follows:

- Each list item clause must have a mappable type.

Cross References

- **`device`** clause, see Section 13.2
- **`target update`** construct, see Section 13.9.
- Array sections, see Section 3.2.4.
- Array shaping, see Section 3.2.3.
- **`from`** clause, see Section 5.9.2
- Iterators, see Section 3.2.5.
- **`to`** clause, see Section 5.9.1
- User-defined mappers, see Section 5.8.10.

5.9.1 `to` Clause

Name: **`to`**

Properties: unique

Arguments:

Name	Type	Properties
locator-list	List containing locator list item	default

Modifiers:

Name	Modifies	Type	Properties
expectation	Generic	**Keyword: `present`**	default
mapper	Generic	Complex modifier: Keyword:**`mapper`** Arguments: Name:*mapper-identifier* Type: Identifier of type mapper Properties: default	unique
item-modifier	Generic	iterator modifier	default

Directives:

`target update`

Semantics

The **`to`** clause is a data motion clause that specifies movement to the targeted devices from the encountering device so the corresponding list items are the assigned list items and the compatible map types are **`to`** and **`tofrom`**.

Cross References

- **`target update`** construct, see Section 13.9.
- Iterators, see Section 3.2.5.

5.9.2 `from` Clause

Name: **`from`**

Properties: unique

Arguments:

Name	Type	Properties
locator-list	List containing locator list item	default

Modifiers:

Name	Modifies	Type	Properties
expectation	Generic	**Keyword: `present`**	default
mapper	Generic	Complex modifier: Keyword:**`mapper`** Arguments: Name:*mapper-identifier* Type: Identifier of type mapper Properties: default	unique
item-modifier	Generic	iterator modifier	default

Directives:

`target update`

Semantics

The **`from`** clause is a data motion clause that specifies movement from the targeted devices to the encountering device so the original list items are the assigned list items and the compatible map types are **`from`** and **`tofrom`**.

Cross References

- **`target update`** construct, see Section 13.9.
- Iterators, see Section 3.2.5.

5.10 `enter` Clause

Name:	Properties:
`enter`	default

Arguments:

Name	Type	Properties
list	List containing extended list item	default

Directives:

`begin declare target`, **`declare target`**

Additional information: The *clause-name* **`to`** may be used as a synonym for the *clause-name* **`enter`**. This use has been deprecated.

Semantics

The **`enter`** clause is a data-mapping clause.

C / C++

If a function appears in an **enter** clause in the same compilation unit in which the definition of the function occurs then a device-specific version of the function is created for all devices to which the directive applies.

If a variable appears in an **enter** clause in the same compilation unit in which the definition of the variable occurs then the original list item is allocated a corresponding list item in the device data environment of all devices to which the directive applies.

C / C++

Fortran

If a procedure appears in an **enter** clause in the same compilation unit in which the definition of the procedure occurs then a device-specific version of the procedure is created for all devices to which the directive applies.

If a variable that is host associated appears in an **enter** clause then the original list item is allocated a corresponding list item in the device data environment of all devices to which the directive applies.

Fortran

If a variable appears in an **enter** clause then the corresponding list item in the device data environment of each device to which the directive applies is initialized once, in the manner specified by the program, but at an unspecified point in the program prior to the first reference to that list item. The list item is never removed from those device data environments as if its reference count was initialized to positive infinity.

Cross References

- **begin declare target** directive, see Section 7.8.2.
- **declare target** directive, see Section 7.8.1.

6 Memory Management

This chapter defines directives, clauses and related concepts for managing memory used by OpenMP programs.

6.1 Memory Spaces

OpenMP memory spaces represent storage resources where variables can be stored and retrieved. Table 6.1 shows the list of predefined memory spaces. The selection of a given memory space expresses an intent to use storage with certain traits for the allocations. The actual storage resources that each memory space represents are implementation defined.

TABLE 6.1: Predefined Memory Spaces

Memory space name	Storage selection intent
omp_default_mem_space	Represents the system default storage
omp_large_cap_mem_space	Represents storage with large capacity
omp_const_mem_space	Represents storage optimized for variables with constant values
omp_high_bw_mem_space	Represents storage with high bandwidth
omp_low_lat_mem_space	Represents storage with low latency

Variables allocated in the **omp_const_mem_space** memory space may be initialized through the **firstprivate** clause or with compile time constants for static and constant variables. Implementation-defined mechanisms to provide the constant value of these variables may also be supported.

Restrictions

Restrictions to OpenMP memory spaces are as follows:

- Variables in the **omp_const_mem_space** memory space may not be written.

Cross References

- **omp_init_allocator** routine, see Section 18.13.2.

6.2 Memory Allocators

OpenMP memory allocators can be used by a program to make allocation requests. When a memory allocator receives a request to allocate storage of a certain size, an allocation of logically consecutive *memory* in the resources of its associated memory space of at least the size that was requested will be returned if possible. This allocation will not overlap with any other existing allocation from an OpenMP memory allocator.

The behavior of the allocation process can be affected by the allocator traits that the user specifies. Table 6.2 shows the allowed allocator traits, their possible values and the default value of each trait.

TABLE 6.2: Allocator Traits

Allocator trait	Allowed values	Default value
sync_hint	**contended**, **uncontended**, **serialized**, **private**	**contended**
alignment	A positive integer value that is a power of 2	1 byte
access	**all**, **cgroup**, **pteam**, **thread**	**all**
pool_size	Positive integer value	Implementation defined
fallback	**default_mem_fb**, **null_fb**, **abort_fb**, **allocator_fb**	**default_mem_fb**
fb_data	an allocator handle	(none)
pinned	**true**, **false**	**false**
partition	**environment**, **nearest**, **blocked**, **interleaved**	**environment**

The **sync_hint** trait describes the expected manner in which multiple threads may use the allocator. The values and their descriptions are:

- **contended**: high contention is expected on the allocator; that is, many threads are expected to request allocations simultaneously.
- **uncontended**: low contention is expected on the allocator; that is, few threads are expected to request allocations simultaneously.
- **serialized**: only one thread at a time will request allocations with the allocator. Requesting two allocations simultaneously when specifying **serialized** results in unspecified behavior.
- **private**: the same thread will request allocations with the allocator every time. Requesting an allocation from different threads, simultaneously or not, when specifying **private** results in

unspecified behavior.

Allocated memory will be byte aligned to at least the value specified for the **alignment** trait of the allocator. Some directives and API routines can specify additional requirements on alignment beyond those described in this section.

Memory allocated by allocators with the **access** trait defined to be **all** must be accessible by all threads in the device where the allocation was requested. Memory allocated by allocators with the **access** trait defined to be **cgroup** will be memory accessible by all threads in the same contention group as the thread that requested the allocation. Attempts to access the memory returned by an allocator with the **access** trait defined to be **cgroup** from a thread that is not part of the same contention group as the thread that allocated the memory result in unspecified behavior. Memory allocated by allocators with the **access** trait defined to be **pteam** will be memory accessible by all threads that bind to the same **parallel** region of the thread that requested the allocation. Attempts to access the memory returned by an allocator with the **access** trait defined to be **pteam** from a thread that does not bind to the same **parallel** region as the thread that allocated the memory result in unspecified behavior. Memory allocated by allocators with the **access** trait defined to be **thread** will be memory accessible by the thread that requested the allocation. Attempts to access the memory returned by an allocator with the **access** trait defined to be **thread** from a thread other than the one that allocated the memory result in unspecified behavior.

The total amount of storage in bytes that an allocator can use is limited by the **pool_size** trait. For allocators with the **access** trait defined to be **all**, this limit refers to allocations from all threads that access the allocator. For allocators with the **access** trait defined to be **cgroup**, this limit refers to allocations from threads that access the allocator from the same contention group. For allocators with the **access** trait defined to be **pteam**, this limit refers to allocations from threads that access the allocator from the same parallel team. For allocators with the **access** trait defined to be **thread**, this limit refers to allocations from each thread that accesses the allocator. Requests that would result in using more storage than **pool_size** will not be fulfilled by the allocator.

The **fallback** trait specifies how the allocator behaves when it cannot fulfill an allocation request. If the **fallback** trait is set to **null_fb**, the allocator returns the value zero if it fails to allocate the memory. If the **fallback** trait is set to **abort_fb**, the behavior is as if an **error** directive for which *sev-level* is **fatal** and *action-time* is **execution** is encountered if the allocation fails. If the **fallback** trait is set to **allocator_fb** then when an allocation fails the request will be delegated to the allocator specified in the **fb_data** trait. If the **fallback** trait is set to **default_mem_fb** then when an allocation fails another allocation will be tried in **omp_default_mem_space**, which assumes all allocator traits to be set to their default values except for **fallback** trait, which will be set to **null_fb**.

Allocators with the **pinned** trait defined to be **true** ensure that their allocations remain in the same storage resource at the same location for their entire lifetime.

The **partition** trait describes the partitioning of allocated memory over the storage resources represented by the memory space associated with the allocator. The partitioning will be done in parts with a minimum size that is implementation defined. The values are:

- **environment**: the placement of allocated memory is determined by the execution environment;
- **nearest**: allocated memory is placed in the storage resource that is nearest to the thread that requests the allocation;
- **blocked**: allocated memory is partitioned into parts of approximately the same size with at most one part per storage resource; and
- **interleaved**: allocated memory parts are distributed in a round-robin fashion across the storage resources.

Table 6.3 shows the list of predefined memory allocators and their associated memory spaces. The predefined memory allocators have default values for their allocator traits unless otherwise specified.

TABLE 6.3: Predefined Allocators

Allocator name	Associated memory space	Non-default trait values
omp_default_mem_alloc	**omp_default_mem_space**	**fallback:null_fb**
omp_large_cap_mem_alloc	**omp_large_cap_mem_space**	(none)
omp_const_mem_alloc	**omp_const_mem_space**	(none)
omp_high_bw_mem_alloc	**omp_high_bw_mem_space**	(none)
omp_low_lat_mem_alloc	**omp_low_lat_mem_space**	(none)
omp_cgroup_mem_alloc	Implementation defined	**access:cgroup**
omp_pteam_mem_alloc	Implementation defined	**access:pteam**
omp_thread_mem_alloc	Implementation defined	**access:thread**

Fortran

If any operation of the base language causes a reallocation of a variable that is allocated with a memory allocator then that memory allocator will be used to deallocate the current memory and to allocate the new memory. For allocated allocatable components of such variables, the allocator that will be used for the deallocation and allocation is unspecified.

Fortran

Cross References

- **OMP_ALLOCATOR** environment variable, see Section 21.5.1.
- **omp_destroy_allocator** routine, see Section 18.13.3.
- **omp_get_default_allocator** routine, see Section 18.13.5.
- **omp_init_allocator** routine, see Section 18.13.2.
- **omp_set_default_allocator** routine, see Section 18.13.4.

6.3 aligned Clause

Name:	**Properties:**
aligned	unique, post-modified

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Modifiers:

Name	Modifies	Type	Properties
alignment	*list*	OpenMP integer expression	unique, ultimate, region invariant

Directives:

declare simd, **simd**

Semantics

The *alignment* modifier specifies the alignment that the program ensures related to the list items. If the *alignment* modifier is not specified, implementation-defined default alignments for SIMD instructions on the target platforms are assumed.

C / C++

The **aligned** clause declares that the object to which each list item points is aligned to the number of bytes expressed in *alignment*.

C / C++

Fortran

The **aligned** clause declares that the target of each list item is aligned to the number of bytes expressed in *alignment*.

Fortran

Restrictions

Restrictions to the **aligned** clause are as follows:

C

- The type of list items must be array or pointer.

C

C++

- The type of list items must be array, pointer, reference to array, or reference to pointer.

C++

Fortran

- Each list item must have **C_PTR** or Cray pointer type or have the **POINTER** or **ALLOCATABLE** attribute. Cray pointer support has been deprecated.
- If a list item has the **ALLOCATABLE** attribute, the allocation status must be allocated.
- If a list item has the **POINTER** attribute, the association status must be associated.
- If the type of a list item is either **C_PTR** or Cray pointer, it must be defined. Cray pointer support has been deprecated.

Fortran

Cross References

- **declare simd** directive, see Section 7.7.
- **simd** construct, see Section 10.4

6.4 align Clause

Name:	**Properties:**
align	unique

Arguments:

Name	Type	Properties
alignment	Expression of type integer	constant, positive

Directives:

allocate

Semantics

The **align** clause is used to specify the byte alignment to use for allocations associated with the construct on which the clause appears. Specifically, each allocation is byte aligned to at least the maximum of the value to which *alignment* evaluates and the alignment required by the base language for the type of the variable that is allocated. On constructs on which the clause may appear, if it is not specified then the effect is as if it was specified with the **alignment** trait of the allocator being used for the allocation.

Restrictions

Restrictions to the **align** clause are as follows:

- *alignment* must evaluate to a power of two.

Cross References

- **allocate** clause, see Section 6.7.
- **allocate** directive, see Section 6.6.
- Memory allocators, see Section 6.2.

6.5 allocator Clause

Name:	**Properties:**
allocator	unique

Arguments:

Name	Type	Properties
allocator	Expression of type allocator_handle	default

Directives:

allocate

Semantics

The **allocator** clause specifies the memory allocator to be used for allocations associated with the construct on which the clause appears. Specifically, the allocator to which *allocator* evaluates is used for the allocations. On constructs on which the clause may appear, if it is not specified then the effect is as if it was specified with the value of the *def-allocator-var* ICV.

Cross References

- **allocate** clause, see Section 6.7.
- **allocate** directive, see Section 6.6.
- Memory allocators, see Section 6.2.

6.6 allocate Directive

Name: allocate	**Association:** none
Category: declarative	**Properties:** default

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Clauses:
align, **allocator**

Semantics

The **allocate** directive specifies how to allocate the specified variables.

The storage for each list item that appears in the **allocate** directive is provided an allocation through the memory allocator as determined by the **allocator** clause with an alignment as determined by the **align** clause. The scope of this allocation is that of the list item in the base language. At the end of the scope for a given list item the memory allocator used to allocate that list item deallocates the storage.

For allocations that arise from this directive the **null_fb** value of the fallback allocator trait behaves as if the **abort_fb** had been specified.

Restrictions

Restrictions to the **allocate** directive are as follows:

- A variable that is part of another variable (as an array element or a structure element) cannot appear in a **allocate** directive.
- An **allocate** directive must appear in the same scope as the declarations of each of its list items and must follow all such declarations.
- A declared variable may appear as a list item in at most one **allocate** directive in a given compilation unit.
- **allocate** directives that appear in a **target** region must specify an **allocator** clause unless a **requires** directive with the **dynamic_allocators** clause is present in the same compilation unit.

C / C++

- If a list item has static storage duration, the **allocator** clause must be specified and the *allocator* expression in the clause must be a constant expression that evaluates to one of the predefined memory allocator values.
- A variable that is declared in a namespace or global scope may only appear as a list item in an **allocate** directive if an **allocate** directive that lists the variable follows a declaration that defines the variable and if all **allocate** directives that list the variable specify the same allocator.

C / C++

C

- After a list item has been allocated, the scope that contains the **allocate** directive must not end abnormally, such as through a call to the **longjmp** function.

C

C++

- After a list item has been allocated, the scope that contains the **allocate** directive must not end abnormally, such as through a call to the **longjmp** function, other than through C++ exceptions.
- A variable that has a reference type may not appear as a list item in an **allocate** directive.

C++

Fortran

- A list item that is specified in an **allocate** directive must not have the **ALLOCATABLE** or **POINTER** attribute.
- If a list item has the **SAVE** attribute, either explicitly or implicitly, or is a common block name then the **allocator** clause must be specified and only predefined memory allocator parameters can be used in the clause.
- A variable that is part of a common block may not be specified as a list item in an **allocate** directive, except implicitly via the named common block.
- A named common block may appear as a list item in at most one **allocate** directive in a given compilation unit.
- If a named common block appears as a list item in an **allocate** directive, it must appear as a list item in an **allocate** directive that specifies the same allocator in every compilation unit in which the common block is used.
- An associate name may not appear as a list item in an **allocate** directive.

Fortran

Cross References

- **align** clause, see Section 6.4.
- **allocator** clause, see Section 6.5.
- *def-allocator-var* ICV, see Section 2.1.
- Memory allocators, see Section 6.2.
- **omp_allocator_handle_t** and **omp_allocator_handle_kind**, see Section 18.13.1.

6.7 `allocate` Clause

Name: `allocate` | **Properties:** default

Arguments:

Name	Type	Properties
list	List containing variable list item	default

Modifiers:

Name	Modifies	Type	Properties
allocator-modifier	*list*	expression of OpenMP allocator_handle type	unique, ultimate
align-modifier	*list*	Complex modifier: Keyword:**`align`** Arguments: Name:*alignment* Type: Expression of type integer Properties: default	unique

Directives:

`allocators`, **`distribute`**, **`do`**, **`for`**, **`parallel`**, **`scope`**, **`sections`**, **`single`**, **`target`**, **`task`**, **`taskloop`**, **`teams`**

Semantics

The **`allocate`** clause specifies the memory allocator to be used to obtain storage for a list of variables. The storage for the list items that appear in the clause is provided through the memory allocator as determined by the *allocator-modifier*, and the provided storage has an alignment as determined by the *align-modifier*. If a list item in the clause also appears in a data-sharing attribute clause on the same directive that privatizes the list item, the allocated storage will be for the new list item. The *align-modifier* has identical syntax and semantics to the **`align`** clause. The *allocator-modifier* has identical semantics to the **`allocator`** clause; it may also be specified as a complex modifier with identical syntax to that clause. If the syntax of the **`allocator`** clause is used for the *allocator-modifier* then its position is unconstrained.

For allocations that arise from this clause the **`null_fb`** value of the fallback allocator trait behaves as if the **`abort_fb`** had been specified.

Restrictions

Restrictions to the **`allocate`** clause are as follows:

- For any list item that is specified in the **`allocate`** clause on a directive other than the **`allocators`** directive, a data-sharing attribute clause that may create a private copy of that list item must be specified on the same directive.

- For **task**, **taskloop** or **target** directives, allocation requests to memory allocators with the trait **access** set to **thread** result in unspecified behavior.
- **allocate** clauses that appear on a **target** construct or on constructs in a **target** region must specify an *allocator* expression unless a **requires** directive with the **dynamic_allocators** clause is present in the same compilation unit.

Cross References

- **align** clause, see Section 6.4.
- **allocator** clause, see Section 6.5.
- *def-allocator-var* ICV, see Section 2.1.
- List Item Privatization, see Section 5.3.
- Memory allocators, see Section 6.2.
- **omp_allocator_handle_t** and **omp_allocator_handle_kind**, see Section 18.13.1.

Fortran

6.8 allocators Construct

Name: allocators	**Association:** block (allocator structured block)
Category: executable	**Properties:** default

Clauses:
allocate

Additional information: The **allocators** construct may alternatively be expressed as one or more **allocate** directives that precede the allocator structured block. The syntax of these directives are as described in Section 6.6, except that the *list* directive argument is optional. If a *list* argument is not specified, the effect is as if there is an implicit list consisting of the names of each variable to be allocated in the associated *allocate-stmt* that is not explicitly listed in another **allocate** directive associated with the statement. **allocate** directives are semantically equivalent to an **allocators** directive that specifies OpenMP allocators and the variables to which they apply in one or more **allocate** clauses, and restricted uses of the **allocators** directive imply that equivalent uses of **allocate** directives are also restricted. This alternate syntax has been deprecated.

Semantics

The **allocators** construct specifies that OpenMP memory allocators are used for certain variables that are allocated by the associated *allocate-stmt*. If a variable that is to be allocated appears as a list item in an **allocate** clause on the directive, an OpenMP allocator is used to allocate storage for the variable according to the semantics of the **allocate** clause. If a variable that is to be allocated does not appear as a list item in an **allocate** clause, the allocation is performed according to the base language implementation.

Restrictions

Restrictions to the **allocators** construct are as follows:

- A list item that appears in an **allocate** clause must appear as one of the variables that is allocated by the *allocate-stmt* in the associated allocator structured block.

Additional restrictions to the (deprecated) **allocate** directive when it is associated with an allocator structured block are as follows:

- If a *list* is specified, the directive must be preceded by an executable statement or OpenMP construct.
- If multiple **allocate** directives are associated with an allocator structured block, at most one directive may specify no list items.

Cross References

- *def-allocator-var* ICV, see Section 2.1.
- Memory allocators, see Section 6.2.
- OpenMP allocator structured blocks, see Section 4.3.1.
- **allocate** clause, see Section 6.7.
- **allocate** directive, see Section 6.6.

Fortran

6.9 uses_allocators Clause

Name:	Properties:
uses_allocators	default

Arguments:

Name	Type	Properties
allocator	Variable of type allocator_handle	default

Modifiers:

Name	Modifies	Type	Properties
mem-space	Generic	Complex modifier: Keyword:**`memspace`** Arguments: Name:*memspace-handle* Type: Variable of type memspace_handle Properties: default	unique
traits-array	Generic	Complex modifier: Keyword:**`traits`** Arguments: Name:*ntraits* Type: Variable of type integer Properties: Name:*traits* Type: Variable of type alloctrait array Properties: ultimate	unique, complex

Directives:
`target`

Additional information: The comma-separated list syntax, in which each list item is a *clause-argument-specification* of the form *allocator[* **`(`***traits***`)`** *]* may also be used for the **`uses allocator`** clause arguments. With this syntax, *traits* must be a constant array with constant values. This syntax has been deprecated.

Semantics

The **`uses_allocators`** clause enables the use of each specified *allocator* in the region associated with the directive on which the clause appears. If *allocator* is a predefined allocator, that predefined allocator will be available for use in the region. If *allocator* is not a predefined allocator, the effect is as if *allocator* is specified on a **`private`** clause. The resulting corresponding item is assigned the result of a call to **`omp_init_allocator`** at the beginning of the associated region with arguments *memspace*, *ntraits*, and *traits*; if *mem-space* is not specified, the effect is as if *memspace* is specified as **`omp_default_mem_space`**. Further, at the end of the associated region, the effect is as if this allocator is destroyed as if by a call to **`omp_destroy_allocator`**.

Restrictions

- If *allocator* is a predefined allocator, no modifiers may be specified.
- If *allocator* is not a predefined allocator, *traits-array* must be specified.

- *allocator* cannot appear in other data-sharing attribute clauses or data-mapping attribute clauses on the same construct.

Cross References

- **`allocate`** clause, see Section 6.7.
- **`allocate`** directive, see Section 6.6.
- **`target`** construct, see Section 13.8.
- Memory allocators, see Section 6.2.

7 Variant Directives

This chapter defines directives and related concepts to support the seamless adaption of programs to OpenMP contexts.

7.1 OpenMP Context

At any point in a program, an OpenMP context exists that defines traits that describe the active OpenMP constructs, the execution devices, functionality supported by the implementation and available dynamic values. The traits are grouped into trait sets. The following trait sets exist: *construct*, *device*, *target_device*, *implementation* and *dynamic*. Traits are categorized as name-list traits, clause-list traits, non-property traits and extension traits. This categorization determines the syntax that is used to match the trait, as defined in Section 7.2.

The *construct* set is composed of the directive names, each being a trait, of all enclosing constructs at that point in the program up to a **target** construct. Combined and composite constructs are added to the set as distinct constructs in the same nesting order specified by the original construct. Whether the **dispatch** construct is added to the *construct* set is implementation defined. If it is added, it will only be added for the *target-call* of the associated code. The set is ordered by nesting level in ascending order. Specifically, the ordering of the set of constructs is $c_1, \ldots, c_N$, where c_1 is the construct at the outermost nesting level and c_N is the construct at the innermost nesting level. In addition, if the point in the program is not enclosed by a **target** construct, the following rules are applied in order:

1. For procedures with a **declare simd** directive, the *simd* trait is added to the beginning of the set as c_1 for any generated SIMD versions so the total size of the set is increased by one.
2. For procedures that are determined to be function variants by a declare variant directive, the selectors $c_1, \ldots, c_M$ of the **construct** selector set are added in the same order to the beginning of the set as $c_1, \ldots, c_M$ so the total size of the set is increased by M.

C / C++

3. For functions that are declared in a code region that is delimited by a declare target directive and its paired end directive, the target trait is added to the beginning of the set as c_1 for any target variants that result from the directive so the total size of the set is increased by one.

C / C++

Fortran

3. If a **declare target** directive appears in the specification part of a procedure or in the specification part of a procedure interface body, the target trait is added to the beginning of the set as c_1 for any target variants that result from the directive so the total size of the set is increased by one.

Fortran

The *simd* trait is a clause-list trait that is defined with properties that match the clauses accepted by the **declare simd** directive with the same name and semantics. The *simd* trait defines at least the *simdlen* property and one of the *inbranch* or *notinbranch* properties. Traits in the *construct* set other than *simd* are non-property traits.

The *device* set includes traits that define the characteristics of the device being targeted by the compiler at that point in the program. For each *target device* that the implementation supports, a *target_device* set exists that defines the characteristics of that device. At least the following traits must be defined for the *device* and all *target_device* sets:

- The *kind(kind-name-list)* trait specifies the general kind of the device. The following *kind-name* values are defined:
  - *host*, which specifies that the device is the host device;
  - *nohost*, which specifies that the devices is not the host device; and
  - the values defined in the *OpenMP Additional Definitions* document.
- The *isa(isa-name-list)* trait specifies the Instruction Set Architectures supported by the device. The accepted *isa-name* values are implementation defined.
- The *arch(arch-name-list)* trait specifies the architectures supported by the device. The accepted *arch-name* values are implementation defined.

The *kind*, *isa* and *arch* traits in the *device* and *target_device* sets are name-list traits.

Additionally, the *target_device* set defines the following trait:

- The *device_num* trait specifies the *device number* of the device.

The *implementation* set includes traits that describe the functionality supported by the OpenMP implementation at that point in the program. At least the following traits can be defined:

- The *vendor(vendor-name-list)* trait, which specifies the vendor identifiers of the implementation. OpenMP defined values for *vendor-name* are defined in the *OpenMP Additional Definitions* document.
- The *extension(extension-name-list)* trait, which specifies vendor specific extensions to the OpenMP specification. The accepted *extension-name* values are implementation defined.

- A trait with a name that is identical to the name of any clause that was supplied to the **requires** directive prior to the program point. Such traits other than the *atomic_default_mem_order* trait are non-property traits. The presence of these traits has been deprecated.
- A *requires(requires-clause-list)* trait, which is a clause-list trait for which the properties are the clauses that have been supplied to the **requires** directive prior to the program point as well as implementation defined implicit requirements.

The *vendor* and *extension* traits in the *implementation* set are name-list traits.

Implementations can define additional traits in the *device*, *target_device* and *implementation* sets; these traits are extension traits.

The *dynamic* trait set includes traits that define the dynamic properties of a program at a point in its execution. The *data state* trait in the *dynamic* trait set refers to the complete data state of the program that may be accessed at runtime.

7.2 Context Selectors

Context selectors are used to define the properties that can match an OpenMP context. OpenMP defines different sets of selectors, each containing different selectors.

The syntax for a context selector is *context-selector-specification* as described in the following grammar:

```
context-selector-specification:
    trait-set-selector[,trait-set-selector[,...]]

trait-set-selector:
    trait-set-selector-name={trait-selector[, trait-selector[, ...]]}

trait-selector:
    trait-selector-name[([trait-score: ] trait-property[, trait-property[, ...]])]

trait-property:
    trait-property-name
    or
    trait-property-clause
    or
    trait-property-expression
    or
    trait-property-extension

trait-property-clause:
    clause
```

```
trait-property-name:
    identifier
    or
    string-literal

trait-property-expression
    scalar-expression (for C/C++)
    or
    scalar-logical-expression (for Fortran)
    or
    scalar-integer-expression (for Fortran)

trait-score:
    score(score-expression)

trait-property-extension:
    trait-property-name
    or
    identifier(trait-property-extension[, trait-property-extension[, ...]])
    or
    constant integer expression
```

For trait selectors that correspond to name-list traits, each *trait-property* should be *trait-property-name* and for any value that is a valid identifier both the identifier and the corresponding string literal (for C/C++) and the corresponding *char-literal-constant* (for Fortran) representation are considered representations of the same value.

For trait selectors that correspond to clause-list traits, each *trait-property* should be *trait-property-clause*. The syntax is the same as for the matching OpenMP clause.

The **construct** selector set defines the *construct* traits that should be active in the OpenMP context. Each selector that can be defined in the **construct** set is the *directive-name* of a context-matching construct. Each *trait-property* of the **simd** selector is a *trait-property-clause*. The syntax is the same as for a valid clause of the **declare simd** directive and the restrictions on the clauses from that directive apply. The **construct** selector is an ordered list $c_1, \ldots, c_N$.

The **device** and **implementation** selector sets define the traits that should be active in the corresponding trait set of the OpenMP context. The **target_device** selector set defines the traits that should be active in the *target_device* trait set for the device that the specified **device_num** selector identifies. The same traits that are defined in the corresponding traits sets can be used as selectors with the same properties. The **kind** selector of the **device** and **target_device** selector sets can also specify the value **any**, which is as if no **kind** selector was specified. If a **device_num** selector does not appear in the **target_device** selector set then a **device_num** selector that specifies the value of the *default-device-var* ICV is implied. For the **device_num** selector of the **target_device** selector set, a single *trait-property-expression* must be specified. For the **atomic_default_mem_order** selector of the **implementation** set, a single *trait-property* must be specified as an identifier equal to one of the valid arguments to the **atomic_default_mem_order** clause on the **requires** directive. For the **requires** selector of the **implementation** set, each *trait-property* is a *trait-property-clause*. The syntax is the same as for a valid clause of the **requires** directive and the restrictions on the clauses from that directive apply.

The **user** selector set defines the **condition** selector that provides additional user-defined conditions.

The **condition** selector contains a single *trait-property-expression* that must evaluate to *true* for the selector to be true.

Any non-constant expression that is evaluated to determine the suitability of a variant is evaluated according to the *data state* trait in the *dynamic* trait set of the OpenMP context.

The **user** selector set is dynamic if the **condition** selector is present and the expression in the **condition** selector is not a constant expression; otherwise, it is static.

All parts of a context selector define the static part of the context selector except the following parts, which define the dynamic part of a context selector:

- Its **user** selector set if it is dynamic; and
- Its **target_device** selector set.

For the **match** clause of a **declare variant** directive, any argument of the base function that is referenced in an expression that appears in the context selector is treated as a reference to the expression that is passed into that argument at the call to the base function. Otherwise, a variable or procedure reference in an expression that appears in a context selector is a reference to the variable or procedure of that name that is visible at the location of the directive on which the selector appears.

C++

Each occurrence of the **this** pointer in an expression in a context selector that appears in the **match** clause of a **declare variant** directive is treated as an expression that is the address of the object on which the associated base function is invoked.

C++

Implementations can allow further selectors to be specified. Each specified *trait-property* for these implementation-defined selectors should be *trait-property-extension*. Implementations can ignore specified selectors that are not those described in this section.

Restrictions

Restrictions to context selectors are as follows:

- Each *trait-property* can only be specified once in a *trait-selector* other than the **construct** selector set.
- Each *trait-set-selector-name* can only be specified once.
- Each *trait-selector-name* can only be specified once.
- A *trait-score* cannot be specified in traits from the **construct**, **device** or **target_device** *trait-selector-sets*.
- A *score-expression* must be a non-negative constant integer expression.
- The expression of a **device_num** trait must evaluate to a non-negative integer value that is less than or equal to the value of **omp_get_num_devices()**.
- A variable or procedure that is referenced in an expression that appears in a context selector must be visible at the location of the directive on which the selector appears unless the directive is a **declare variant** directive and the variable is an argument of the associated base function.
- If *trait-property* **any** is specified in the **kind** *trait-selector* of the **device** or **target_device** selector set, no other *trait-property* may be specified in the same selector.
- For a *trait-selector* that corresponds to a name-list trait, at least one *trait-property* must be specified.
- For a *trait-selector* that corresponds to a non-property trait, no *trait-property* may be specified.
- For the **requires** selector of the **implementation** selector set, at least one *trait-property* must be specified.

7.3 Matching and Scoring Context Selectors

A given context selector is compatible with a given OpenMP context if the following conditions are satisfied:

- All selectors in the **user** set of the context selector are true;
- All traits and trait properties that are defined by selectors in the **target_device** set of the context selector are active in the *target_device* trait set for the device that is identified by the **device_num** selector;
- All traits and trait properties that are defined by selectors in the **construct**, **device** and **implementation** sets of the context selector are active in the corresponding trait sets of the OpenMP context;
- For each selector in the context selector, its properties are a subset of the properties of the corresponding trait of the OpenMP context;

- Selectors in the **construct** set of the context selector appear in the same relative order as their corresponding traits in the *construct* trait set of the OpenMP context; and
- No specified implementation-defined selector is ignored by the implementation.

Some properties of the **simd** selector have special rules to match the properties of the *simd* trait:

- The **simdlen(**N**)** property of the selector matches the *simdlen(M)* trait of the OpenMP context if $M\%N$ equals zero; and
- The **aligned(***list:N***)** property of the selector matches the *aligned(list:M)* trait of the OpenMP context if $N\%M$ equals zero.

Among compatible context selectors, a score is computed using the following algorithm:

1. Each trait selector for which the corresponding trait appears in the *construct* trait set in the OpenMP context is given the value 2^{p-1} where p is the position of the corresponding trait, c_p, in the context *construct* trait set; if the traits that correspond to the **construct** selector set appear multiple times in the OpenMP context, the highest valued subset of context traits that contains all selectors in the same order are used;
2. The **kind**, **arch**, and **isa** selectors, if specified, are given the values 2^l, 2^{l+1} and 2^{l+2}, respectively, where l is the number of traits in the *construct* set;
3. Trait selectors for which a *trait-score* is specified are given the value specified by the *trait-score score-expression*;
4. The values given to any additional selectors allowed by the implementation are implementation defined;
5. Other selectors are given a value of zero; and
6. A context selector that is a strict subset of another context selector has a score of zero. For other context selectors, the final score is the sum of the values of all specified selectors plus 1.

7.4 Metadirectives

A metadirective is a directive that can specify multiple directive variants of which one may be conditionally selected to replace the metadirective based on the enclosing OpenMP context. A metadirective is replaced by a **nothing** directive or one of the directive variants specified by the **when** clauses or the **otherwise** clause. If no **otherwise** clause is specified the effect is as if one was specified without an associated directive variant.

The OpenMP context for a given metadirective is defined according to Section 7.1. The order of clauses that appear on a metadirective is significant and **otherwise** must be the last clause specified on a metadirective.

Replacement candidates are ordered according to the following rules in decreasing precedence:

- A candidate is before another one if the score associated with the context selector of the corresponding **when** clause is higher.
- A candidate that was explicitly specified is before one that was implicitly specified.
- Candidates are ordered according to the order in which they lexically appear on the metadirective.

The list of dynamic replacement candidates is the prefix of the sorted list of replacement candidates up to and including the first candidate for which the corresponding **when** clause has a static context selector.

The first dynamic replacement candidate for which the corresponding **when** clause has a compatible context selector, according to the matching rules defined in Section 7.3, replaces the metadirective.

Restrictions

Restrictions to metadirectives are as follows:

- Replacement of the metadirective with the directive variant associated with any of the dynamic replacement candidates must result in a conforming OpenMP program.
- Insertion of user code at the location of a metadirective must be allowed if the first dynamic replacement candidate does not have a static context selector.
- All items must be executable directives if the first dynamic replacement candidate does not have a static context selector.

Fortran

- A metadirective that appears in the specification part of a subprogram must follow all *variant-generating* declarative directives that appear in the same specification part.

Fortran

7.4.1 when Clause

Name:	**Properties:**
when	default

Arguments:

Name	Type	Properties
directive-variant	directive-specification	default

Modifiers:

Name	Modifies	Type	Properties
context-selector	*directive-variant*	An OpenMP context-selector-specification	default

Directives:

begin metadirective, **metadirective**

Semantics

The directive variant specified by a **when** clause is a candidate to replace the metadirective on which the clause is specified if the static part of the corresponding context selector is compatible with the OpenMP context according to the matching rules defined in Section 7.3. If a **when** clause does not explicitly specify a directive variant it implicitly specifies a **nothing** directive as the directive variant.

Expressions that appear in the context selector of a **when** clause are evaluated if no prior dynamic replacement candidate has a compatible context selector, and the number of times each expression is evaluated is implementation defined. All variables referenced by these expressions are considered to be referenced by the metadirective.

A directive variant that is associated with a **when** clause can only affect the program if the directive variant is a dynamic replacement candidate.

Restrictions

Restrictions to the **when** clause are as follows:

- *directive-variant* must not specify a metadirective.

C / C++

- *directive-variant* must not specify a **begin declare variant** directive.

C / C++

- *context-selector* must not specify any properties for the **simd** selector.

Cross References

- Metadirectives, see Section 7.4.
- Context selectors, see Section 7.2.
- **begin metadirective**, see Section 7.4.4.
- **metadirective**, see Section 7.4.3.

7.4.2 `otherwise` Clause

Name:	**Properties:**
otherwise	unique, ultimate

Arguments:

Name	Type	Properties
directive-variant	directive-specification	default

Directives:

begin metadirective, **metadirective**

Additional information: The *clause-name* **default** may be used as a synonym for the *clause-name* **otherwise**. This use has been deprecated.

Semantics

The **default** clause is treated as a **when** clause with the specified directive variant, if any, and an always compatible static context selector that has a score lower than the scores associated with any other clause.

Restrictions

Restrictions to the **otherwise** clause are as follows:

- *directive-variant* must not specify a metadirective.

C / C++

- *directive-variant* must not specify a **begin declare variant** directive.

C / C++

Cross References

- Metadirectives, see Section 7.4.
- **begin metadirective**, see Section 7.4.4.
- **metadirective**, see Section 7.4.3.
- **when** Clause, see Section 7.4.1.

7.4.3 metadirective

Name: metadirective **Category:** meta	**Association:** none **Properties:** default

Clauses:

otherwise, **when**

Semantics

The **metadirective** specifies metadirective semantics.

Cross References

- Metadirectives, see Section 7.4.
- **otherwise** Clause, see Section 7.4.2.
- **when** Clause, see Section 7.4.1.

7.4.4 begin metadirective

Name: begin metadirective **Category:** meta	**Association:** delimited **Properties:** default

Clauses:

otherwise, **when**

Semantics

The **begin metadirective** is a metadirective for which the specified directive variants other than the **nothing** directive must accept a paired **end** *directive*. For any directive variant that is selected to replace the **begin metadirective** directive, the **end metadirective** directive is implicitly replaced by its paired **end** directive to demarcate the statements that are affected by or are associated with the directive variant. If the **nothing** directive is selected to replace the **begin metadirective** directive, the paired **end metadirective** is ignored.

Restrictions

The restrictions to **begin metadirective** are as follows:

- Any *directive-variant* that is specified by a **when** or **otherwise** clause must be an OpenMP directive that has a paired **end** directive or must be the **nothing** directive.

Cross References

- Metadirectives, see Section 7.4.
- **otherwise** Clause, see Section 7.4.2.
- **when** Clause, see Section 7.4.1.

7.5 Declare Variant Directives

Declare variant directives declare *base functions* to have the specified function variant. The context selector in the **match** clause is associated with the variant.

The OpenMP context for a direct call to a given base function is defined according to Section 7.1. If a declare variant directive for the base function is visible at the call site and the static part of the context selector that is associated with the declared function variant is compatible with the OpenMP context of the call according to the matching rules defined in Section 7.3 then the variant is a replacement candidate to be called instead of the base function. Replacement candidates are ordered in decreasing order of the score associated with the context selector. If two replacement candidates have the same score then their order is implementation defined.

The list of dynamic replacement candidates is the prefix of the sorted list of replacement candidates up to and including the first candidate for which the corresponding context selector is static.

The first dynamic replacement candidate for which the corresponding context selector is compatible, according to the matching rules defined in Section 7.3, is called instead of the base function. If no compatible candidate exists then the base function is called.

Expressions that appear in the context selector of a **match** clause are evaluated if no prior dynamic replacement candidate has a compatible context selector, and the number of times each expression is evaluated is implementation defined. All variables referenced by these expressions are considered to be referenced at the call site.

C++

For calls to **`constexpr`** base functions that are evaluated in constant expressions, whether any variant replacement occurs is implementation defined.

C++

For indirect function calls that can be determined to call a particular base function, whether any variant replacement occurs is unspecified.

Any differences that the specific OpenMP context requires in the prototype of the variant from the base function prototype are implementation defined.

Different declare variant directives may be specified for different declarations of the same base function.

Restrictions

Restrictions to declare variant directives are as follows:

- Calling functions that a declare variant directive determined to be a function variant directly in an OpenMP context that is different from the one that the **`construct`** selector set of the context selector specifies is non-conforming.
- If a function is determined to be a function variant through more than one declare variant directive then the **`construct`** selector set of their context selectors must be the same.
- A function determined to be a function variant may not be specified as a base function in another declare variant directive.
- An **`adjust_args`** clause or **`append_args`** clause can only be specified if the **`dispatch`** selector of the **`construct`** selector set appears in the **`match`** clause.

C / C++

- The type of the function variant must be compatible with the type of the base function after the implementation-defined transformation for its OpenMP context.

C / C++

C++

- Declare variant directives cannot be specified for virtual, defaulted or deleted functions.
- Declare variant directives cannot be specified for constructors or destructors.
- Declare variant directives cannot be specified for immediate functions.
- The function that a declare variant directive determined to be a function variant may not be an immediate function.

C++

Cross References

- Context Selectors, see Section 7.2.
- OpenMP Context Specification, see Section 7.1.
- **begin declare variant** directive, see Section 7.5.5.
- **declare variant** directive, see Section 7.5.4.

7.5.1 match Clause

Name:	Properties:
match	unique

Arguments:

Name	Type	Properties
directive-variant	directive-specification	default

Modifiers:

Name	Modifies	Type	Properties
context-selector	*directive-variant*	An OpenMP context-selector-specification	default

Directives:

begin metadirective, **metadirective**

Semantics

The **match** clause specifies the *context-selector* to use to determine if a specified variant function is a replacement candidate for the specified base function in a given context.

Restrictions

Restrictions to the **match** clause are as follows:

- All variables that are referenced in an expression that appears in the context selector of a **match** clause must be accessible at a call site to the base function according to the base language rules.

Cross References

- Declare variant directives, see Section 7.5.
- Context selectors, see Section 7.2.
- **begin declare variant** directive, see Section 7.5.5.
- **declare variant** directive, see Section 7.5.4.

7.5.2 `adjust_args` Clause

Name: `adjust_args` **Properties:** default

Arguments:

Name	Type	Properties
parameter-list	List containing parameter list item	default

Modifiers:

Name	Modifies	Type	Properties
adjust-op	*parameter-list*	**Keyword:** **`need_device_ptr`**, **`nothing`**	default

Directives:

`declare variant`

Semantics

The **`adjust_args`** clause specifies how to adjust the arguments of the base function when a specified variant function is selected for replacement. For each **`adjust_args`** clause that is present on the selected variant the adjustment operation specified by *adjust-op* is applied to each argument specified in the clause before being passed to the selected variant. If the *adjust-op* modifier is **`nothing`**, the argument is passed to the selected variant without being modified.

If the *adjust-op* modifier is **`need_device_ptr`**, the arguments are converted to corresponding device pointers of the default device. If an argument has the *is_device_ptr* property in its *interoperability requirement set* then the argument is not adjusted. Otherwise, the argument is converted in the same manner that a **`use_device_ptr`** clause on a **`target data`** construct converts its pointer list items into device pointers. If the argument cannot be converted into a device pointer then *NULL* is passed as the argument.

Restrictions

Fortran

- Each argument that appears in a **`need_device_ptr`** *adjust-op* must be of type **`C_PTR`** in the dummy argument declaration of the variant function.

Fortran

Cross References

- Declare variant directives, see Section 7.5.
- **`declare variant`** directive, see Section 7.5.4.

7.5.3 `append_args` Clause

Name:	Properties:
append_args	unique

Arguments:

Name	Type	Properties
append-op-list	List containing interop type	default

Directives:

declare variant

Semantics

The **append_args** clause specifies additional arguments to pass in the call when a specified variant function is selected for replacement. The arguments are constructed according to any specified *append-op* modifiers and are passed in the same order in which they are specified in the **append_args** clause.

For each member of *append-op-list*, the **interop** operation constructs an argument of that OpenMP **interop** type from the *interoperability requirement set* of the encountering task. The argument is constructed as if an **interop** construct with an **init** clause of *interop-types* was specified. If the *interoperability requirement set* contains one or more properties that could be used as clauses for an **interop** construct of *interop-type*, the behavior is as if the corresponding clauses would also be part of the aforementioned **interop** construct and those properties are removed from the *interoperability requirement set*.

This argument is destroyed after the call to the selected variant returns, as if an **interop** construct with a **destroy** clause was used with the same clauses that were used to initialize the argument.

Cross References

- Declare variant directives, see Section 7.5.
- Interoperability requirement set, see Section 14.2.
- **declare variant** directive, see Section 7.5.4.
- **interop** construct, see Section 14.1.

7.5.4 `declare variant` Directive

Name: **declare variant** Category: declarative	Association: declaration Properties: default

Arguments: declare_variant (*[base–name:]variant-name*)

Name	Type	Properties
base-name	Identifier of type function	optional
variant-name	Identifier of type function	default

Clauses:

adjust_args, **append_args**, **match**

Semantics

The **`declare variant`** specifies declare variant semantics for a single replacement candidate. *variant-name* identifies the function variant while *base-name* identifies the base function.

C

Any expressions in the **`match`** clause are interpreted as if they appeared in the scope of arguments of the base function.

C

C++

variant-name and any expressions in the **`match`** clause are interpreted as if they appeared at the scope of the trailing return type of the base function.

The function variant is determined by base language standard name lookup rules ([basic.lookup]) of *variant-name* using the argument types at the call site after implementation-defined changes have been made according to the OpenMP context.

C++

Fortran

The procedure to which *base-name* refers is resolved at the location of the directive according to the establishment rules for procedure names in the base language.

Fortran

Restrictions

- If *base-name* is specified, it must match the name used in the associated declaration, if any declaration is associated.

Fortran

- *base-name* must not be a generic name, an entry name, the name of a procedure pointer, a dummy procedure or a statement function.
- If *base-name* is omitted then the **`declare variant`** directive must appear in an interface block or the specification part of a procedure.
- Any **`declare variant`** directive must appear in the specification part of a subroutine subprogram, function subprogram, or interface body to which it applies.
- If the directive is specified for a procedure that is declared via a procedure declaration statement, the *base-name* must be specified.
- The procedure *base-name* must have an accessible explicit interface at the location of the directive.

Fortran

Cross References

- Declare variant directives, see Section 7.5.
- **adjust_args** Clause, see Section 7.5.2.
- **append_args** Clause, see Section 7.5.3.
- **match** Clause, see Section 7.5.1.

C / C++

7.5.5 begin declare variant Directive

Name: **begin declare variant**	Association: delimited (declaration-definition-seq)
Category: declarative	Properties: default

Clauses:

match

Semantics

The **begin declare variant** directive associates the context selector in the **match** clause with each function definition in *declaration-definition-seq*. For the purpose of call resolution, each function definition that appears between a **begin declare variant** directive and its paired **end** directive is a function variant for an assumed base function, with the same name and a compatible prototype, that is declared elsewhere without an associated declare variant directive.

If a declare variant directive appears between a **begin declare variant** directive and its paired **end** directive, the effective context selectors of the outer directive are appended to the context selector of the inner directive to form the effective context selector of the inner directive. If a *trait-set-selector* is present on both directives, the *trait-selector* list of the outer directive is appended to the *trait-selector* list of the inner directive after equivalent *trait-selectors* have been removed from the outer list. Restrictions that apply to explicitly specified context selectors also apply to effective context selectors constructed through this process.

The symbol name of a function definition that appears between a **begin declare variant** directive and its paired **end** directive is determined through the base language rules after the name of the function has been augmented with a string that is determined according to the effective context selector of the **begin declare variant** directive. The symbol names of two definitions of a function are considered to be equal if and only if their effective context selectors are equivalent.

If the context selector of a **begin declare variant** directive contains traits in the *device* or *implementation* set that are known never to be compatible with an OpenMP context during the current compilation, the preprocessed code that follows the **begin declare variant** directive up to its paired **end** directive is elided.

Any expressions in the **match** clause are interpreted at the location of the directive.

Restrictions

The restrictions to **begin declare variant** directive are as follows:

- **match** clause must not contain a **simd** *trait-selector-name*.
- Two **begin declare variant** directives and their paired **end** directives must either encompass disjoint source ranges or be perfectly nested.
- **match** clause must not contain a dynamic context selector that references the **this** pointer.
- If an expression in the context selector that appears in **match** clause references the **this** pointer, the base function must be a non-static member function.

Cross References

- Declare variant directives, see Section 7.5.
- **match** Clause, see Section 7.5.1.

C / C++

7.6 dispatch Construct

Name: dispatch	**Association:** block (function dispatch structured block)
Category: executable	**Properties:** context-matching

Clauses:

depend, **device**, **is_device_ptr**, **nocontext**, **novariants**, **nowait**

Binding

The binding task set for a **dispatch** region is the generating task. The **dispatch** region binds to the region of the generating task.

Semantics

The **dispatch** construct controls whether variant substitution occurs for a given call.

Properties added to the *interoperability requirement set* can be removed by the effect of other directives (see Section 14.2) before the **dispatch** region is executed. If one or more **depend** clauses are present on the **dispatch** construct, they are added as *depend* properties of the *interoperability requirement set*. If a **nowait** clause is present on the **dispatch** construct the *nowait* property is added to the *interoperability requirement set*. For each list item specified in an **is_device_ptr** clause, an *is_device_ptr* property for that list item is added to the *interoperability requirement set*.

If the *interoperability requirement set* contains one or more *depend* properties, the behavior is as if those properties were applied to a **taskwait** construct as **depend** clauses that is executed before the **dispatch** region is executed.

The presence of the **nowait** property in the *interoperability requirement set* has no effect on the **dispatch** construct.

If the **device** clause is present, the value of the *default-device-var* ICV is set to the value of the expression in the clause on entry to the **dispatch** region and is restored to its previous value at the end of the region.

Cross References

- **declare variant** directive, see Section 7.5.
- Interoperability requirement set, see Section 14.2.
- OpenMP function dispatch structured blocks, see Section 4.3.2.
- **depend** clause, see Section 15.9.5.
- **is_device_ptr** clause, see Section 5.8.3.
- **nocontext** clause, see Section 7.6.2.
- **novariants** clause, see Section 7.6.1.
- **nowait** clause, see Section 15.6.

7.6.1 novariants Clause

Name:	**Properties:**
novariants	unique

Arguments:

Name	Type	Properties
do-not-use-variant	Expression of type logical	default

Directives:

dispatch

Semantics

If *do-not-use-variant* evaluates to *true*, no function variant is selected for the *target-call* of the **dispatch** region associated with the **novariants** clause even if one would be selected normally. The use of a variable in *do-not-use-variant* causes an implicit reference to the variable in all enclosing constructs. *do-not-use-variant* is evaluated in the enclosing context.

Cross References

- **dispatch** construct, see Section 7.6.

7.6.2 nocontext Clause

Name:	**Properties:**
nocontext	unique

Arguments:

Name	Type	Properties
do-not-update-context	Expression of type logical	default

Directives:
dispatch

Semantics

If *do-not-update-context* evaluates to *true*, the construct on which the **nocontext** clause appears is not added to the *construct* set of the OpenMP context. The use of a variable in *do-not-update-context* causes an implicit reference to the variable in all enclosing constructs. *do-not-update-context* is evaluated in the enclosing context.

Cross References

- **dispatch** construct, see Section 7.6.

7.7 declare simd Directive

Name: declare simd	**Association:** declaration
Category: declarative	**Properties:** default

Arguments: declare_simd*[* **(***proc-name***)** *]*

Name	Type	Properties
proc-name	Identifier of type function	optional

Clause groups:
branch

Clauses:
aligned, **linear**, **simdlen**, **uniform**

Semantics

The association of one or more **declare simd** directives with a function declaration or definition enables the creation of corresponding SIMD versions of the associated function that can be used to process multiple arguments from a single invocation in a SIMD loop concurrently.

If a SIMD version is created and the **simdlen** clause is not specified, the number of concurrent arguments for the function is implementation defined.

For purposes of the **linear** clause, any integer-typed parameter that is specified in a **uniform** clause on the directive is considered to be constant and so may be used in *linear-step*.

C / C++

The expressions that appear in the clauses of each directive are evaluated in the scope of the arguments of the function declaration or definition.

C / C++

C++

The special *this* pointer can be used as if it was one of the arguments to the function in any of the **linear**, **aligned**, or **uniform** clauses.

C++

Restrictions

Restrictions to the **declare simd** directive are as follows:

- If *base-name* is specified, it must match the name used in the associated declaration, if any declaration is associated.
- The function or subroutine body must be a structured block.
- The execution of the function or subroutine, when called from a SIMD loop, cannot result in the execution of an OpenMP construct except for an **ordered** construct with the **simd** clause or an **atomic** construct.
- The execution of the function or subroutine cannot have any side effects that would alter its execution for concurrent iterations of a SIMD chunk.
- A program that branches into or out of the function is non-conforming.

C / C++

- If the function has any declarations, then the **declare simd** directive for any declaration that has one must be equivalent to the one specified for the definition. Otherwise, the result is unspecified.
- The function cannot contain calls to the **longjmp** or **setjmp** functions.

C / C++

C++

- The function cannot contain any calls to **throw**.

C++

Fortran

- *proc-name* must not be a generic name, procedure pointer, or entry name.
- If *proc-name* is omitted, the **declare simd** directive must appear in the specification part of a subroutine subprogram or a function subprogram for which creation of the SIMD versions is enabled.
- Any **declare simd** directive must appear in the specification part of a subroutine subprogram, function subprogram, or interface body to which it applies.
- If a **declare simd** directive is specified in an interface block for a procedure, it must match a **declare simd** directive in the definition of the procedure.
- If a procedure is declared via a procedure declaration statement, the procedure *proc-name* should appear in the same specification.
- If a **declare simd** directive is specified for a procedure name with explicit interface and a **declare simd** directive is also specified for the definition of the procedure then the two **declare simd** directives must match. Otherwise the result is unspecified.
- Procedure pointers may not be used to access versions created by the **declare simd** directive.

Fortran

Cross References

- **aligned** clause, see Section 6.3.
- *branch* clauses, see Section 7.7.1.
- **linear** clause, see Section 5.4.6.
- **simdlen** clause, see Section 10.4.3.
- **uniform** clause, see Section 5.4.7.
- **reduction** clause, see Section 5.5.9.

7.7.1 *branch* Clauses

Clause group: branch

Properties: unique, inarguable, fully exclusive	**Members:** **inbranch**, **notinbranch**

Semantics

The *branch* clause grouping defines a set of clauses that indicate if a function can be assumed to be or not to be encountered in a branch. The **inbranch** clause specifies that the function will always be called from inside a conditional statement of the calling context. The **notinbranch** clause specifies that the function will never be called from inside a conditional statement of the calling context. If neither clause is specified, then the function may or may not be called from inside a conditional statement of the calling context.

Cross References

- **declare simd** directive, see Section 7.7.

7.8 Declare Target Directives

Declare target directives apply to procedures and/or variables to ensure that they can be executed or accessed on a device. Variables are mapped for all device executions, or for specific device executions through a **link** clause. An implementation may generate different versions of a procedure to be used for **target** regions that execute on different devices. Whether the same version is generated for different devices, or whether a version that is called in a **target** region differs from the version that is called outside a **target** region, is implementation defined.

To facilitate device usage, OpenMP defines rules that implicitly specify declare target directives for procedures and variables. The remainder of this section defines those rules as well as restrictions that apply to all declare target directives.

If a variable with static storage duration is declared in a device routine then the named variable is treated as if it had appeared in an **enter** clause on a declare target directive.

In the following, a non-host declare target directive is one that does not specify a **device_type** clause with **host**. Further, a reverse-offload region is a region that is associated with a **target** construct that specifies a **device** clause with the **ancestor** *device-modifier*.

C / C++

If a function is referenced outside of any reverse-offload region in a function that appears as a list item in an **enter** clause on a non-host declare target directive then the name of the referenced function is treated as if it had appeared in an **enter** clause on a declare target directive.

If a variable with static storage duration or a function (except *lambda* for C++) is referenced in the initializer expression list of a variable with static storage duration that appears as a list item in an **enter** clause on a declare target directive then the name of the referenced variable or function is treated as if it had appeared in an **enter** clause on a declare target directive.

C / C++

Fortran

If a procedure is referenced outside of any reverse-offload region in a procedure that appears as a list item in an **enter** clause on a non-host **declare target** directive then the name of the referenced procedure is treated as if it had appeared in an **enter** clause on a **declare target** directive.

If a **declare target** directive has a **device_type** clause then any enclosed internal procedures cannot contain any **declare target** directives. The enclosing **device_type** clause implicitly applies to internal procedures.

Fortran

Execution Model Events

The *target-global-data-op* event occurs when an original variable is associated with a corresponding variable on a device as a result of a declare target directive; the event occurs before the first access to the corresponding variable.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target_data_op** callback, or a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_beginend** as its endpoint argument for each occurrence of a *target-global-data-op* event in that thread. These callbacks have type signature **ompt_callback_target_data_op_t** or **ompt_callback_target_data_op_emi_t**, respectively.

Restrictions

Restrictions to the declare target directive are as follows:

- A threadprivate variable cannot appear in the directive.
- A variable declared in the directive must have a mappable type.
- A variable declared in the directive must have static storage duration.
- The same list item must not explicitly appear in both a **enter** clause on one declare target directive and a **link** clause on another declare target directive.
- If the directive has a clause, it must contain at least one **enter** clause or at least one **link** clause.
- A variable for which **nohost** is specified may not appear in a **link** clause.
- If a variable appears in a **enter** clause on the declare target directive, its initializer must not refer to a variable that appears in a **link** clause on a declare target directive.

C++

- A variable that is part of another variable (as an array element or a structure element) cannot appear as a list item in an **enter** or **link** clause on the directive.

C++

Cross References

- **ompt_callback_target_data_op_t** or **ompt_callback_target_data_op_emi_t** callback type, see Section 19.5.2.25.
- **begin declare target** directive, see Section 7.8.2.
- **declare target** directive, see Section 7.8.1.
- **enter** clause, see Section 5.10.
- **link** clause, see Section 5.8.7.

- **target** construct, see Section 13.8.
- **target data** construct, see Section 13.5.

7.8.1 declare target Directive

Name: **declare target** Category: declarative	Association: none Properties: device, declare target

Arguments: declare_target (*extended-list*)

Name	Type	Properties
extended-list	List containing extended list item	optional

Clauses:
device_type, **enter**, **indirect**, **link**

Semantics

The **declare target** directive is a declare target directive. If the *extended-list* argument is specified, the effect is as if an **enter** clause was specified with the *extended-list* as its argument.

Fortran

If a **declare target** does not have any clauses and does not have an *extended-list* then an implicit **enter** clause with one item is formed from the name of the enclosing subroutine subprogram, function subprogram or interface body to which it applies.

Fortran

Restrictions

Restrictions to the **declare target** directive are as follows:

- If the *extended-list* argument is specified, no clauses may be specified.

Fortran

- If a list item is a procedure name, it must not be a generic name, procedure pointer, entry name, or statement function name.
- If no clauses are specified or if a **device_type** clause is specified, the directive must appear in a specification part of a subroutine subprogram, function subprogram or interface body.
- If a list item is a procedure name, the directive must be in the specification part of that subroutine or function subprogram or in the specification part of that subroutine or function in an interface body.
- If an extended list item is a variable name, the directive must appear in the specification part of a subroutine subprogram, function subprogram, program or module.
- If the directive is specified in an interface block for a procedure, it must match a **declare target** directive in the definition of the procedure, including the **device_type** clause if present.

- If an external procedure is a type-bound procedure of a derived type and the directive is specified in the definition of the external procedure, it must appear in the interface block that is accessible to the derived-type definition.
- If any procedure is declared via a procedure declaration statement that is not in the type-bound procedure part of a derived-type definition, any **declare target** with the procedure name must appear in the same specification part.
- The directive must appear in the declaration section of a scoping unit in which the common block or variable is declared.
- If a **declare target** directive that specifies a common block name appears in one program unit, then such a directive must also appear in every other program unit that contains a **COMMON** statement that specifies the same name, after the last such **COMMON** statement in the program unit.
- If a list item is declared with the **BIND** attribute, the corresponding C entities must also be specified in a **declare target** directive in the C program.
- A variable can only appear in a **declare target** directive in the scope in which it is declared. It must not be an element of a common block or appear in an **EQUIVALENCE** statement.
- A variable that appears in a **declare target** directive must be declared in the Fortran scope of a module or have the **SAVE** attribute, either explicitly or implicitly.

Fortran

Cross References

- Declare target directives, see Section 7.8.
- **device_type** clause, see Section 13.1.
- **enter** clause, see Section 5.10.
- **indirect** clause, see Section 7.8.3.
- **link** clause, see Section 5.8.7.

C / C++

7.8.2 begin declare target Directive

Name: **begin declare target** **Category:** declarative	**Association:** delimited **Properties:** device, declare target

Clauses:
device_type, **indirect**

Additional information: The directive name **declare target** may be used as a synonym to **begin declare target** if no clauses are specified. This syntax has been deprecated.

Semantics

The **begin declare target** directive is a declare target directive. The directive and its paired **end** directive form a delimited code region that defines an implicit *extended-list*. The implicit *extended-list* consists of the variable names of any variable declarations at file or namespace scope that appear in the delimited code region and of the function names of any function declarations at file, namespace or class scope that appear in the delimited code region. The implicit *extended-list* is converted to an implicit **enter** clause.

The delimited code region may contain declare target directives. If a **device_type** clause is present on the contained declare target directive, then its argument determines which versions are made available. If a list item appears both in an implicit and explicit list, the explicit list determines which versions are made available.

Restrictions

Restrictions to the **begin declare target** directive are as follows:

C++

- The function names of overloaded functions or template functions may only be specified within an implicit *extended-list*.
- If a *lambda declaration and definition* appears between a **begin declare target** directive and the paired **end** directive, all variables that are captured by the *lambda* expression must also appear in an **enter** clause.
- A module *export* or *import* statement cannot appear between a declare target directive and the paired **end** directive.

C++

Cross References

- Declare target directives, see Section 7.8.
- **device_type** clause, see Section 13.1.
- **enter** clause, see Section 5.10.
- **indirect** clause, see Section 7.8.3.

C / C++

7.8.3 `indirect` Clause

Name:	**Properties:**
`indirect`	unique

Arguments:

Name	Type	Properties
invoked-by-fptr	Expression of type logical	constant

Directives:
`begin declare target`, **`declare target`**

Semantics

If *invoked-by-fptr* evaluates to true, any procedures that appear in an **`enter`** clause on the directive on which the **`indirect`** clause is specified may be called with an indirect device invocation. If the *invoked-by-fptr* does not evaluate to true, any procedures that appear in an **`enter`** clause on the directive may not be called with an indirect device invocation. Unless otherwise specified by an **`indirect`** clause, procedures may not be called with an indirect device invocation. If the **`indirect`** clause is specified and *invoked-by-fptr* is not specified, the effect of the clause is as if *invoked-by-fptr* evaluates to true.

C / C++

If a function appears in the **`enter`** clause of a **`begin declare target`** directive and in the **`enter`** clause of a declare target directive that is contained in the delimited code region of the **`begin declare target`** directive, and if an **`indirect`** clause appears on both directives, then the **`indirect`** clause on the **`begin declare target`** directive has no effect for that function.

C / C++

Restrictions

Restrictions to the **`indirect`** clause are as follows:

- If *invoked-by-fptr* evaluates to true, a **`device_type`** clause must not appear on the same directive unless it specifies **`any`**. for its *device-type-description*.

Cross References

- **`begin declare target`** directive, see Section 7.8.2.
- **`declare target`** directive, see Section 7.8.1.

8 Informational and Utility Directives

An informational directive conveys information about code properties to the compiler while a utility directive facilitates interactions with the compiler or supports code readability. Utility directives can be informational but may be executable as determined by the **at** clause.

8.1 at Clause

Name:	**Properties:**
at	unique

Arguments:

Name	Type	Properties
action-time	**Keyword: compilation, execution**	default

Directives:
error

Semantics

The **at** clause determines when the implementation performs an action that is associated with a utility directive. If *action-time* is **compilation**, the action is performed during compilation if the directive appears in a declarative context or in an executable context that is reachable at runtime. If *action-time* is **compilation** and the directive appears in an executable context that is not reachable at runtime, the action may or may not be performed. If *action-time* is **execution**, the action is performed during program execution when a thread encounters the directive. If the **at** clause is not specified, the effect is as if *action-time* is **compilation**.

Cross References

- **error** directiver, see Section 8.5.

8.2 requires Directive

Name: requires	**Association:** none
Category: informational	**Properties:** default

Clause groups:
requirement

Semantics

The **requires** directive specifies features that an implementation must support for correct execution. The behavior that a requirement clause specifies may override the normal behavior specified elsewhere in this document. Whether an implementation supports the feature that a given requirement clause specifies is implementation defined. The **requires** directive specifies requirements for the execution of all code in the current compilation unit.

> Note – Use of this directive makes code less portable. Users should be aware that not all devices or implementations support all requirements.

The clauses of a **requires** directive are added to the *requires* trait in the OpenMP context for all program points that follow the directive.

Restrictions

The restrictions to the **requires** directive are as follows:

- All **requires** directives in the same compilation unit that specify the **atomic_default_mem_order** requirement must specify the same parameter.
- Any **requires** directive that specifies a **reverse_offload**, **unified_address**, or **unified_shared_memory** requirement must appear lexically before any device constructs or device routines.
- A **requires** directive may not appear lexically after a context selector in which any clause of the **requires** directive is used.
- Either all compilation units of a program that contain declare target directives, device constructs or device routines or none of them must specify a **requires** directive that specifies the **reverse_offload**, **unified_address** or **unified_shared_memory** requirement.
- A **requires** directive that specifies the **atomic_default_mem_order** requirement must not appear lexically after any **atomic** construct on which *memory-order-clause* is not specified.

C

- The **requires** directive may only appear at file scope.

C

C++

- The **requires** directive may only appear at file or namespace scope.

C++

Fortran

- The **requires** directive must appear in the specification part of a *program unit*, after any **USE** statement, any **IMPORT** statement, and any **IMPLICIT** statement, unless the directive appears by referencing a module and each clause already appeared with the same parameters in the specification part of the *program unit*.

Fortran

8.2.1 *requirement* Clauses

Clause group: requirement

Properties: unique	**Members:** **atomic_default_mem_order**, **dynamic_allocators**, **reverse_offload**, **unified_address**, **unified_shared_memory**

Semantics

The *requirement* clause grouping defines a set of clauses that indicate the requirement that a program requires the implementation to support. Other than **atomic_default_mem_order**, the members of the set are inarguable.

If an implementation supports a given *requirement* clause then the use of that clause on a **requires** directive will cause the implementation to ensure the enforcement of a guarantee represented by the specific member of the clause grouping. If the implementation does not support the requirement then it must report an error.

The **reverse_offload** clause requires an implementation to guarantee that if a **target** construct specifies a **device** clause in which the **ancestor** modifier appears, the **target** region can execute on the parent device of an enclosing **target** region.

The **unified_address** clause requires an implementation to guarantees that all devices accessible through OpenMP API routines and directives use a unified address space. In this address space, a pointer will always refer to the same location in memory from all devices accessible through OpenMP. Any OpenMP mechanism that returns a device pointer is guaranteed to return a device address that supports pointer arithmetic, and the **is_device_ptr** clause is not necessary to obtain device addresses from device pointers for use inside **target** regions. Host pointers may be passed as device pointer arguments to device memory routines and device pointers may be passed as host pointer arguments to device memory routines. Non-host devices may still have discrete memories and dereferencing a device pointer on the host device or a host pointer on a non-host device remains unspecified behavior. Memory local to a specific execution context may be exempt from the **unified_address** requirement, following the restrictions of locality to a given execution context, thread or contention group.

The **unified_shared_memory** clause implies the **unified_address** requirement, inheriting all of its behaviors. The implementation must also guarantee that storage locations in memory are accessible to threads on all available devices that the implementation supports, except for memory that is local to a specific execution context as defined in the description of **unified_address** above. Every device address that refers to storage allocated through OpenMP device memory routines is a valid host pointer that may be dereferenced.

The **unified_shared_memory** clause makes **map** clauses optional on **target** constructs and declare target directives optional for variables with static storage duration that are accessed inside functions to which a declare target directive is applied. Scalar variables are still firstprivate by default when referenced inside **target** constructs. Values stored into memory by one device may not be visible to another device until those two devices synchronize with each other or both devices synchronize with the host.

The **dynamic_allocators** clause removes certain restrictions on the use of memory allocators in **target** regions. Specifically, allocators may be used in a **target** region without specifying the **uses_allocators** clause on the corresponding **target** construct. The implementation must support calls to the **omp_init_allocator** and **omp_destroy_allocator** API routines in **target** regions. Finally, default allocators may be used on **allocate** directives and **allocate** clauses, and in **omp_alloc** API routines in **target** regions.

The **atomic_default_mem_order** clause specifies the default memory ordering behavior for **atomic** constructs that an implementation must provide. The effect is as if its parameter appears as a clause on any **atomic** construct that does not specify a memory order clause.

Cross References

- **requires** directive, see Section 8.2.

8.3 Assumption Directives

Assumption directives provide invariants that specify additional information about the expected properties of the program that can optionally be used to optimize the implementation. If the invariants do not hold at runtime, the behavior is unspecified. An implementation may ignore this information without altering the behavior of the program. Different assumption directive formats facilitate definition of assumptions for a scope that is appropriate to each base language. The scope of a particular format is its *assumption scope* and is defined in the section that defines that format.

8.3.1 *assumption* Clauses

Clause group: assumption

Properties:	**Members:** **absent**, **contains**, **holds**, **no_openmp**, **no_openmp_routines**, **no_parallelism**

Semantics

The *assumption* clause grouping defines a set of clauses that indicate the assumptions that a program ensures the implementation can exploit. Other than **absent**, **contains** and **holds**, the members of the set are inarguable and unique.

The **no_openmp** clause guarantees that no OpenMP related code is executed in the assumption scope. The **no_openmp_routines** clause guarantees that no explicit OpenMP runtime library calls are executed in the assumption scope. The **no_parallelism** clause guarantees that no OpenMP tasks (explicit or implicit) will be generated and that no SIMD constructs will be executed in the assumption scope.

C++

The **no_openmp** clause also guarantees that no thread will throw an exception in the assumption scope if it is contained in a region that arises from an exception-aborting directive.

C++

The **absent** and **contains** clauses accept a *directive-name* list that may match a construct that is encountered within the assumption scope. An encountered construct matches the directive name if it or (if it is a combined or composite construct) one of its leaf constructs has the same *directive-name* as one of the members of the list. The **absent** clause specifies that the application guarantees that no constructs that match a listed directive name are encountered in the assumption scope. The **contains** clause specifies that constructs that match the listed directive names are likely to be encountered in the assumption scope.

When the **holds** clause appears on an assumption directive, the application guarantees that the listed expression evaluates to *true* in the assumption scope. The effect of the clause does not include an observable evaluation of the expression.

Restrictions

The restrictions to *assumption* clauses are as follows:

- A *directive-name* list member cannot specify a combined or composite directive.
- A *directive-name* list member cannot specify a directive that is not associated with the execution of user or implementation code, i.e., a **nothing** directive, a declarative directive, a metadirective, or a loop transformation directive.

8.3.2 assumes Directive

Name: assumes **Category:** informational	**Association:** none **Properties:** default

Clause groups:
assumption

Semantics

The assumption scope of the **assumes** directive is the code executed and reached from the current compilation unit.

Restrictions

The restrictions to *assumes* directive are as follows:

C

- The **assumes** directive may only appear at file scope.

C

C++

- The **assumes** directive may only appear at file or namespace scope.

C++

Fortran

- The **assumes** directive may only appear in the specification part of a module or subprogram, after any **USE** statement, any **IMPORT** statement, and any **IMPLICIT** statement.

Fortran

8.3.3 assume Directive

Name: **assume** **Category:** informational	**Association:** block **Properties:** default

Clause groups:

assumption

Semantics

The assumption scope of the **assume** directive is the code executed in the corresponding region or in any region that is nested in the corresponding region.

C / C++

8.3.4 begin assumes Directive

Name: **begin assumes** **Category:** informational	**Association:** delimited **Properties:** default

Clause groups:

assumption

Semantics

The assumption scope of the **begin assumes** directive is the code that is executed and reached from any of the declared functions in the delimited code region.

C / C++

8.4 `nothing` Directive

Name: `nothing` **Category:** utility	**Association:** none **Properties:** default

Semantics

The **`nothing`** directive has no effect on the execution of the OpenMP program.

Cross References

- Metadirectives, see Section 7.4.

8.5 `error` Directive

Name: `error` **Category:** utility	**Association:** none **Properties:** default

Clauses:

`at`, **`message`**, **`severity`**

Semantics

The **`error`** directive instructs the compiler or runtime to perform an error action. The error action displays an implementation-defined message. The **`severity`** clause determines whether the error action is abortive following the display of the message. If *sev-level* is **`fatal`** and **`action-time`** is **`compilation`**, the message is displayed and compilation of the current compilation unit is aborted. If *sev-level* is **`fatal`** and *action-time* is **`execution`**, the message is displayed and program execution is aborted.

Execution Model Events

The *runtime-error* event occurs when a thread encounters an **`error`** directive for which the **`at`** clause specifies **`execution`**.

Tool Callbacks

A thread dispatches a registered **`ompt_callback_error`** callback for each occurrence of a *runtime-error* event in the context of the encountering task. This callback has the type signature **`ompt_callback_error_t`**.

Cross References

- **`at`** clause, see Section 8.1.
- **`ompt_callback_error_t`**, see Section 19.5.2.30.
- **`message`** clause, see Section 8.5.2.
- **`severity`** clause, see Section 8.5.1.

8.5.1 `severity` Clause

Name:	**Properties:**
`severity`	unique

Arguments:

Name	Type	Properties
sev-level	**Keyword: `fatal`, `warning`**	default

Directives:

`error`

Semantics

The **`severity`** clause determines the action that the implementation performs. If *sev-level* is **`warning`**, the implementation takes no action besides displaying the message that is associated with the directive. if *sev-level* is **`fatal`**, the implementation performs the abortive action associated with the directive on which the clause appears. If no **`severity`** clause is specified then the effect is as if *sev-level* is **`fatal`**.

Restrictions

- *hint-expr* must evaluate to a valid synchronization hint.

Cross References

- **`error`** directiver, see Section 8.5.

8.5.2 `message` Clause

Name:	**Properties:**
`message`	unique

Arguments:

Name	Type	Properties
msg-string	Expression of type string	default

Directives:

`error`

Semantics

The **`message`** clause specifies that *msg-string* is included in the implementation-defined message that is associated with the directive on which the clause appears.

Cross References

- **`error`** directiver, see Section 8.5.

9 Loop Transformation Constructs

A loop transformation construct replaces itself, including its associated loop nest, with a structured block that may be another loop nest. If the loop transformation construct is nested inside another loop nest, its replacement becomes part of that loop nest and therefore its generated loops may become associated with another loop-associated directive that forms an enclosing construct. A loop transformation construct that is closely nested within another loop transformation construct applies before the enclosing loop transformation construct.

The associated loop nest of a loop transformation construct must have *canonical loop nest form* (see Section 4.4.1). All generated loops have canonical loop nest form, unless otherwise specified. Loop iteration variables of generated loops are always private in the enclosing parallelism-generating construct.

Cross References

- Canonical loop nest form, see Section 4.4.1.

9.1 `tile` Construct

Name: `tile`	**Association:** loop
Category: executable	**Properties:** default

Clauses:
`sizes`

Semantics

The **`tile`** construct tiles the outer n loops of the associated loop nest, where n is the number of items in *size-list*, which consists of items $s_1, \ldots, s_n$. Let $\ell_1, \ldots \ell_n$ be the associated loops, from outermost to innermost, which the construct replaces with a loop nest that consists of $2n$ perfectly nested loops. Let $f_1, \ldots, f_n, t_1, \ldots, t_n$ be the generated loops, from outermost to innermost. The loops $f_1, \ldots, f_n$ are the *floor loops* and the loops $t_1, \ldots, t_n$ are the *tile loops*. The tile loops do not have canonical loop nest form.

Let Ω be the *logical iteration vector space* of the associated loops. For any $(\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n$, define the set of iterations $\{(i_1, \ldots, i_n) \in \Omega \mid \forall k \in \{1, \ldots, n\} : s_k\alpha_k \le i_k < s_k\alpha_k + s_k\}$ to be tile $T_{\alpha_1,\ldots,\alpha_n}$ and $F = \{T_{\alpha_1,\ldots,\alpha_n} \mid T_{\alpha_1,\ldots,\alpha_n} \neq \emptyset\}$ to be the set of tiles with at least one iteration. Tiles that contain $\prod_{k=1}^{n} s_k$ iterations are complete tiles. Otherwise, they are partial tiles.

The floor loops iterate over all tiles $\{T_{\alpha_1,\ldots,\alpha_n} \in F\}$ in lexicographic order with respect to their indices $(\alpha_1, \ldots, \alpha_n)$ and the tile loops iterate over the iterations in $T_{\alpha_1,\ldots,\alpha_n}$ in the lexicographic order of the corresponding iteration vectors. An implementation may reorder the sequential execution of two iterations if at least one is from a partial tile and if their respective logical iteration vectors in *loop-nest* do not have a product order relation.

Restrictions

Restrictions to the **tile** construct are as follows:

- The depth of the associated loop nest must be greater than or equal to n.
- All loops that are associated with the construct must be perfectly nested.
- No loop that is associated with the construct may be a non-rectangular loop.

Cross References

- Canonical loop nest form, see Section 4.4.1.
- Worksharing-loop construct, see Section 11.5.
- **distribute** construct, see Section 11.6.
- **taskloop** construct, see Section 12.6.

9.1.1 sizes Clause

Name:	**Properties:**
sizes	unique, required

Arguments:

Name	Type	Properties
size-list	Expression of type integer	constant, positive

Directives:

tile

Semantics

The **sizes** clause specifies a list of *n* compile-time constant, positive OpenMP integer expressions.

Cross References

- **tile** construct, see Section 9.1.

9.2 `unroll` Construct

Name: `unroll` **Category:** executable	**Association:** loop **Properties:** default

Clauses:
`full`, `partial`

Clause set:

Properties: fully exclusive	**Members:** full, partial

Semantics

The `unroll` construct unrolls the outermost loop of the loop nest according to its specified clause. If no clauses are specified, if and how the loop is unrolled is implementation defined. The `unroll` construct results in a generated loop that has canonical loop nest form if and only if the `partial` clause is specified.

Cross References

- Canonical loop nest form, see Section 4.4.1.

9.2.1 `full` Clause

Name: `full]`	**Properties:** unique

Directives:
`unroll`

Semantics

The `full` clause specifies that the associated loop is *fully unrolled*. The construct is replaced by a structured block that only contains n instances of its loop body, one for each of the n logical iterations of the associated loop and in their logical iteration order.

Restrictions

Restrictions to the `full` clause are as follows:

- The iteration count of the associated loop must be a compile-time constant.

Cross References

- `unroll` construct, see Section 9.2.

9.2.2 **partial** Clause

Name:	**Properties:**
partial]	unique

Arguments:

Name	Type	Properties
unroll-factor	Expression of type integer	optional, constant, positive

Directives:

unroll

Semantics

The **partial** clause specifies that the associated loop is first tiled with a tile size of *unroll-factor*. Then, the generated tile loop is fully unrolled. If the **partial** clause is used without an *unroll-factor* argument then the unroll factor is a positive integer that is implementation defined.

Cross References

- **unroll** construct, see Section 9.2.

10 Parallelism Generation and Control

This chapter defines constructs for generating and controlling parallelism.

10.1 `parallel` Construct

Name: `parallel` **Category:** executable	**Association:** block **Properties:** parallelism-generating, cancellable, thread-limiting, context-matching

Clauses:
`allocate`, **`copyin`**, **`default`**, **`firstprivate`**, **`if`**, **`num_threads`**, **`private`**, **`proc_bind`**, **`reduction`**, **`shared`**

Binding

The binding thread set for a **`parallel`** region is the encountering thread. The encountering thread becomes the primary thread of the new team.

Semantics

When a thread encounters a **`parallel`** construct, a team of threads is created to execute the **`parallel`** region (see Section 10.1.1 for more information about how the number of threads in the team is determined, including the evaluation of the **`if`** and **`num_threads`** clauses). The thread that encountered the **`parallel`** construct becomes the primary thread of the new team, with a thread number of zero for the duration of the new **`parallel`** region. All threads in the new team, including the primary thread, execute the region. Once the team is created, the number of threads in the team remains constant for the duration of that **`parallel`** region.

Within a **`parallel`** region, thread numbers uniquely identify each thread. Thread numbers are consecutive whole numbers ranging from zero for the primary thread up to one less than the number of threads in the team. A thread may obtain its own thread number by a call to the **`omp_get_thread_num`** library routine.

A set of implicit tasks, equal in number to the number of threads in the team, is generated by the encountering thread. The structured block of the **`parallel`** construct determines the code that will be executed in each implicit task. Each task is assigned to a different thread in the team and becomes tied. The task region of the task that the encountering thread is executing is suspended and each thread in the team executes its implicit task. Each thread can execute a path of statements that is different from that of the other threads.

The implementation may cause any thread to suspend execution of its implicit task at a task scheduling point, and to switch to execution of any explicit task generated by any of the threads in the team, before eventually resuming execution of the implicit task (for more details see Section 12).

An implicit barrier occurs at the end of a **parallel** region. After the end of a **parallel** region, only the primary thread of the team resumes execution of the enclosing task region.

If a thread in a team that is executing a **parallel** region encounters another **parallel** directive, it creates a new team, according to the rules in Section 10.1.1, and it becomes the primary thread of that new team.

If execution of a thread terminates while inside a **parallel** region, execution of all threads in all teams terminates. The order of termination of threads is unspecified. All work done by a team prior to any barrier that the team has passed in the program is guaranteed to be complete. The amount of work done by each thread after the last barrier that it passed and before it terminates is unspecified.

Execution Model Events

The *parallel-begin* event occurs in a thread that encounters a **parallel** construct before any implicit task is created for the corresponding **parallel** region.

Upon creation of each implicit task, an *implicit-task-begin* event occurs in the thread that executes the implicit task after the implicit task is fully initialized but before the thread begins to execute the structured block of the **parallel** construct.

If the **parallel** region creates a native thread, a *native-thread-begin* event occurs as the first event in the context of the new thread prior to the *implicit-task-begin* event.

Events associated with implicit barriers occur at the end of a **parallel** region. Section 15.3.2 describes events associated with implicit barriers.

When a thread finishes an implicit task, an *implicit-task-end* event occurs in the thread after events associated with implicit barrier synchronization in the implicit task.

The *parallel-end* event occurs in the thread that encounters the **parallel** construct after the thread executes its *implicit-task-end* event but before the thread resumes execution of the encountering task.

If a native thread is destroyed at the end of a **parallel** region, a *native-thread-end* event occurs in the thread as the last event prior to destruction of the thread.

Tool Callbacks

A thread dispatches a registered **ompt_callback_parallel_begin** callback for each occurrence of a *parallel-begin* event in that thread. The callback occurs in the task that encounters the **parallel** construct. This callback has the type signature **ompt_callback_parallel_begin_t**. In the dispatched callback, **(***flags* **& ompt_parallel_team)** evaluates to *true*.

A thread dispatches a registered **ompt_callback_implicit_task** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *implicit-task-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_implicit_task** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of an *implicit-task-end* event in that thread. The callbacks occur in the context of the implicit task and have type signature **ompt_callback_implicit_task_t**. In the dispatched callback, **(***flags* **& ompt_task_implicit)** evaluates to *true*.

A thread dispatches a registered **ompt_callback_parallel_end** callback for each occurrence of a *parallel-end* event in that thread. The callback occurs in the task that encounters the **parallel** construct. This callback has the type signature **ompt_callback_parallel_end_t**.

A thread dispatches a registered **ompt_callback_thread_begin** callback for the *native-thread-begin* event in that thread. The callback occurs in the context of the thread. The callback has type signature **ompt_callback_thread_begin_t**.

A thread dispatches a registered **ompt_callback_thread_end** callback for the *native-thread-end* event in that thread. The callback occurs in the context of the thread. The callback has type signature **ompt_callback_thread_end_t**.

Cross References

- OpenMP execution model, see Section 1.3.
- **if** clause, see Section 3.4.
- **ompt_callback_implicit_task_t**, see Section 19.5.2.11.
- **ompt_callback_parallel_begin_t**, see Section 19.5.2.3.
- **ompt_callback_parallel_end_t**, see Section 19.5.2.4.
- **ompt_callback_thread_begin_t**, see Section 19.5.2.1.
- **ompt_callback_thread_end_t**, see Section 19.5.2.2.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 19.4.4.11.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **copyin** clause, see Section 5.7.
- **default**, **shared**, **private**, **firstprivate**, and **reduction** clauses, see Section 5.4.
- Determining the number of threads for a parallel region, see Section 10.1.1.
- **allocate** clause, see Section 6.7.
- **num_threads** clause, see Section 10.1.2.
- **omp_get_thread_num** routine, see Section 18.2.4.
- **proc_bind** clause, see Section 10.1.4.

10.1.1 Determining the Number of Threads for a `parallel` Region

When execution encounters a **`parallel`** directive, the value of the **`if`** clause or **`num_threads`** clause (if any) on the directive, the current parallel context, and the values of the *nthreads-var*, *dyn-var*, *thread-limit-var*, and *max-active-levels-var* ICVs are used to determine the number of threads to use in the region.

Using a variable in an **`if`** or **`num_threads`** clause expression of a **`parallel`** construct causes an implicit reference to the variable in all enclosing constructs. The **`if`** clause expression and the **`num_threads`** clause expression are evaluated in the context outside of the **`parallel`** construct, and no ordering of those evaluations is specified. In what order or how many times any side effects of the evaluation of the **`num_threads`** or **`if`** clause expressions occur is also unspecified.

When a thread encounters a **`parallel`** construct, the number of threads is determined according to Algorithm 2.1.

Algorithm 2.1

---

let *ThreadsBusy* be the number of OpenMP threads currently executing in this contention group;

if an **`if`** clause exists

then let *IfClauseValue* be the value of the **`if`** clause expression;

else let *IfClauseValue* = *true*;

if a **`num_threads`** clause exists

then let *ThreadsRequested* be the value of the **`num_threads`** clause expression;

else let *ThreadsRequested* = value of the first element of *nthreads-var*;

let *ThreadsAvailable* = (*thread-limit-var* - *ThreadsBusy* + 1);

if (*IfClauseValue* = *false*)

then number of threads = 1;

else if (*active-levels-var* $\geq$ *max-active-levels-var*)

then number of threads = 1;

else if (*dyn-var* = *true*) **and** (*ThreadsRequested* $\leq$ *ThreadsAvailable*)

then 1 $\leq$ number of threads $\leq$ *ThreadsRequested*;

else if (*dyn-var* = *true*) **and** (*ThreadsRequested* $>$ *ThreadsAvailable*)

then 1 $\leq$ number of threads $\leq$ *ThreadsAvailable*;

else if (*dyn-var* = *false*) **and** (*ThreadsRequested* $\leq$ *ThreadsAvailable*)

then number of threads = *ThreadsRequested*;

else if (*dyn-var* = *false*) **and** (*ThreadsRequested* > *ThreadsAvailable*)

then behavior is implementation defined;

Note – Since the initial value of the *dyn-var* ICV is implementation defined, programs that depend on a specific number of threads for correct execution should explicitly disable dynamic adjustment of the number of threads.

Cross References

- *nthreads-var*, *dyn-var*, *thread-limit-var*, and *max-active-levels-var* ICVs, see Section 2.
- **if** clause, see Section 3.4.
- **parallel** construct, see Section 10.1.
- **num_threads** clause, see Section 10.1.2.

10.1.2 num_threads Clause

Name:	**Properties:**
num_threads	unique

Arguments:

Name	Type	Properties
nthreads	Expression of type integer	positive

Directives:

parallel

Semantics

The **num_threads** clause specifies the desired number of threads to execute a **parallel** region.

Cross References

- **parallel** construct, see Section 10.1.
- Determining the number of threads for a parallel region, see Section 10.1.1.

10.1.3 Controlling OpenMP Thread Affinity

When a thread encounters a **parallel** directive without a **proc_bind** clause, the *bind-var* ICV is used to determine the policy for assigning OpenMP threads to places within the current place partition, that is, within the places listed in the *place-partition-var* ICV for the implicit task of the encountering thread. If the **parallel** directive has a **proc_bind** clause then the binding policy specified by the **proc_bind** clause overrides the policy specified by the first element of the *bind-var* ICV. Once a thread in the team is assigned to a place, the OpenMP implementation should not move it to another place.

The **primary** thread affinity policy instructs the execution environment to assign every thread in the team to the same place as the primary thread. The place partition is not changed by this policy, and each implicit task inherits the *place-partition-var* ICV of the parent implicit task. The **master** thread-affinity policy, which has been deprecated, has identical semantics to the **primary** thread affinity policy.

The **close** thread affinity policy instructs the execution environment to assign the threads in the team to places close to the place of the parent thread. The place partition is not changed by this policy, and each implicit task inherits the *place-partition-var* ICV of the parent implicit task. If T is the number of threads in the team, and P is the number of places in the parent's place partition, then the assignment of threads in the team to places is as follows:

- $T \leq P$: The primary thread executes on the place of the parent thread. The thread with the next smallest thread number executes on the next place in the place partition, and so on, with wrap around with respect to the place partition of the primary thread.
- $T > P$: Each place p will contain S_p threads with consecutive thread numbers where $\lfloor T/P \rfloor \leq S_p \leq \lceil T/P \rceil$. The first S_0 threads (including the primary thread) are assigned to the place of the parent thread. The next S_1 threads are assigned to the next place in the place partition, and so on, with wrap around with respect to the place partition of the primary thread. When P does not divide T evenly, the exact number of threads in a particular place is implementation defined.

The purpose of the **spread** thread affinity policy is to create a sparse distribution for a team of T threads among the P places of the parent's place partition. A sparse distribution is achieved by first subdividing the parent partition into T subpartitions if $T \leq P$, or P subpartitions if $T > P$. Then one thread ($T \leq P$) or a set of threads ($T > P$) is assigned to each subpartition. The *place-partition-var* ICV of each implicit task is set to its subpartition. The subpartitioning is not only a mechanism for achieving a sparse distribution, it also defines a subset of places for a thread to use when creating a nested **parallel** region. The assignment of threads to places is as follows:

- $T \leq P$: The parent thread's place partition is split into T subpartitions, where each subpartition contains $\lfloor P/T \rfloor$ or $\lceil P/T \rceil$ consecutive places. A single thread is assigned to each subpartition. The primary thread executes on the place of the parent thread and is assigned to the subpartition that includes that place. The thread with the next smallest thread number is assigned to the first place in the next subpartition, and so on, with wrap around with respect to the original place partition of the primary thread.

- $T > P$: The parent thread's place partition is split into P subpartitions, each consisting of a single place. Each subpartition is assigned S_p threads with consecutive thread numbers, where $\lfloor T/P \rfloor \leq S_p \leq \lceil T/P \rceil$. The first S_0 threads (including the primary thread) are assigned to the subpartition that contains the place of the parent thread. The next S_1 threads are assigned to the next subpartition, and so on, with wrap around with respect to the original place partition of the primary thread. When P does not divide T evenly, the exact number of threads in a particular subpartition is implementation defined.

The determination of whether the affinity request can be fulfilled is implementation defined. If the affinity request cannot be fulfilled, then the affinity of threads in the team is implementation defined.

Note – Wrap around is needed if the end of a place partition is reached before all thread assignments are done. For example, wrap around may be needed in the case of **`close`** and $T \leq P$, if the primary thread is assigned to a place other than the first place in the place partition. In this case, thread 1 is assigned to the place after the place of the primary thread, thread 2 is assigned to the place after that, and so on. The end of the place partition may be reached before all threads are assigned. In this case, assignment of threads is resumed with the first place in the place partition.

10.1.4 `proc_bind` Clause

Name:	**Properties:**
`proc_bind`	unique

Arguments:

Name	Type	Properties
affinity-policy	**Keyword: `close`**, **`master [deprecated]`**, **`primary`**, **`spread`**	default

Directives:
`parallel`

Semantics

The **`proc_bind`** clause specifies the mapping of OpenMP threads to places within the current place partition, that is, within the places listed in the *place-partition-var* ICV for the implicit task of the encountering thread. The effect of the possible values for *affinity-policy* are described in Section 10.1.3

Cross References

- **`parallel`** construct, see Section 10.1.
- Controlling OpenMP thread affinity, see Section 10.1.3.

10.2 `teams` Construct

Name: `teams` **Category:** executable	**Association:** block **Properties:** parallelism-generating, thread-limiting, context-matching

Clauses:
`allocate`, **`default`**, **`firstprivate`**, **`if`**, **`num_teams`**, **`private`**, **`reduction`**, **`shared`**, **`thread_limit`**

Binding

The binding thread set for a **`teams`** region is the encountering thread.

Semantics

When a thread encounters a **`teams`** construct, a league of teams is created. Each team is an initial team, and the initial thread in each team executes the **`teams`** region. The number of teams created is implementation defined, but it will be greater than or equal to *lower-bound* and less than or equal to *upper-bound* as specified to the **`num_teams`** clause. Once the teams are created, the number of initial teams remains constant for the duration of the **`teams`** region. Within a **`teams`** region, initial team numbers uniquely identify each initial team. Initial team numbers are consecutive whole numbers ranging from zero to one less than the number of initial teams.

If a **`thread_limit`** clause is not present on the **`teams`** construct, but the construct is closely nested inside a **`target`** construct on which the **`thread_limit`** clause is specified, the behavior is as if that **`thread_limit`** clause is also specified for the **`teams`** construct.

On a combined or composite construct that includes **`target`** and **`teams`** constructs, the expressions in **`num_teams`** and **`thread_limit`** clauses are evaluated on the host device on entry to the **`target`** construct.

The place list, given by the *place-partition-var* ICV of the encountering thread, is split into subpartitions in an implementation-defined manner, and each team is assigned to a subpartition by setting the *place-partition-var* of its initial thread to the subpartition.

The **`teams`** construct sets the *default-device-var* ICV for each initial thread to an implementation-defined value.

After the teams have completed execution of the **`teams`** region, the encountering task resumes execution of the enclosing task region.

Execution Model Events

The *teams-begin* event occurs in a thread that encounters a **`teams`** construct before any initial task is created for the corresponding **`teams`** region.

Upon creation of each initial task, an *initial-task-begin* event occurs in the thread that executes the initial task after the initial task is fully initialized but before the thread begins to execute the structured block of the **`teams`** construct.

If the **teams** region creates a native thread, a *native-thread-begin* event occurs as the first event in the context of the new thread prior to the *initial-task-begin* event.

When a thread finishes an initial task, an *initial-task-end* event occurs in the thread.

The *teams-end* event occurs in the thread that encounters the **teams** construct after the thread executes its *initial-task-end* event but before it resumes execution of the encountering task.

If a native thread is destroyed at the end of a **teams** region, a *native-thread-end* event occurs in the thread as the last event prior to destruction of the thread.

Tool Callbacks

A thread dispatches a registered **ompt_callback_parallel_begin** callback for each occurrence of a *teams-begin* event in that thread. The callback occurs in the task that encounters the **teams** construct. This callback has the type signature **ompt_callback_parallel_begin_t**. In the dispatched callback, **(***flags* **& ompt_parallel_league)** evaluates to *true*.

A thread dispatches a registered **ompt_callback_implicit_task** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *initial-task-begin* in that thread. Similarly, a thread dispatches a registered **ompt_callback_implicit_task** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of an *initial-task-end* event in that thread. The callbacks occur in the context of the initial task and have type signature **ompt_callback_implicit_task_t**. In the dispatched callback, **(***flags* **& ompt_task_initial)** evaluates to *true*.

A thread dispatches a registered **ompt_callback_parallel_end** callback for each occurrence of a *teams-end* event in that thread. The callback occurs in the task that encounters the **teams** construct. This callback has the type signature **ompt_callback_parallel_end_t**.

A thread dispatches a registered **ompt_callback_thread_begin** callback for the *native-thread-begin* event in that thread. The callback occurs in the context of the thread. The callback has type signature **ompt_callback_thread_begin_t**.

A thread dispatches a registered **ompt_callback_thread_end** callback for the *native-thread-end* event in that thread. The callback occurs in the context of the thread. The callback has type signature **ompt_callback_thread_end_t**.

Restrictions

Restrictions to the **teams** construct are as follows:

- If a *reduction-modifier* is specified in a **reduction** clause that appears on the directive then the reduction modifer must be **default**.
- A **teams** region must be strictly nested within the implicit parallel region that surrounds the whole OpenMP program or a **target** region. If a **teams** region is nested inside a **target** region, the corresponding **target** construct must not contain any statements, declarations or directives outside of the corresponding **teams** construct.

- **distribute** regions, including any **distribute** regions arising from composite constructs, **parallel** regions, including any **parallel** regions arising from combined constructs, **loop** regions, **omp_get_num_teams()** regions, and **omp_get_team_num()** regions are the only OpenMP regions that may be strictly nested inside the **teams** region.

Cross References

- **ompt_callback_implicit_task_t**, see Section 19.5.2.11.
- **ompt_callback_parallel_begin_t**, see Section 19.5.2.3.
- **ompt_callback_parallel_end_t**, see Section 19.5.2.4.
- **ompt_callback_thread_begin_t**, see Section 19.5.2.1.
- **ompt_callback_thread_end_t**, see Section 19.5.2.2.
- **parallel** construct, see Section 10.1.
- Data-sharing attribute clauses, see Section 5.4.
- **allocate** clause, see Section 6.7.
- **distribute** construct, see Section 11.6.
- **num_teams** clause, see Section 10.2.1.
- **omp_get_num_teams** routine, see Section 18.4.1.
- **omp_get_team_num** routine, see Section 18.4.2.
- **target** construct, see Section 13.8.
- **thread_limit** clause, see Section 13.3.

10.2.1 num_teams Clause

Name:	**Properties:**
num_teams	unique

Arguments:

Name	Type	Properties
upper-bound	Expression of type integer	positive

Modifiers:

Name	Modifies	Type	Properties
lower-bound	Generic	OpenMP integer expression	unique, ultimate, positive

Directives:
parallel

Semantics

The **num_teams** clause specifies the bounds on the number of teams created by the construct on which it appears. *lower-bound* specifies the lower bound and *upper-bound* specifies upper bound on the number of teams requested. If *lower-bound* is not specified, the effect is as if *lower-bound* is specified as equal to *upper-bound*.

If the **num_teams** clause is not specified on a construct then the effect is as if *upper-bound* was specified as follows. If the value of the *nteams-var* ICV is greater than zero, the effect is as if *upper-bound* was specified to an implementation-defined value greater than zero but less than or equal to the value of the *nteams-var* ICV. Otherwise, the effect is as if *upper-bound* was specified as an implementation defined value greater than or equal to one.

Restrictions

- *lower-bound* must be less than or equal to *upper-bound*.

Cross References

- **team** construct, see Section 10.2.

10.3 order Clause

Name:	**Properties:**
order	unique

Arguments:

Name	Type	Properties
ordering	**Keyword: concurrent**	default

Modifiers:

Name	Modifies	Type	Properties
order-modifier	*ordering*	**Keyword: reproducible, unconstrained**	default

Directives:

distribute, **do**, **for**, **loop**, **simd**

Semantics

The **order** clause specifies an *ordering* of execution for the iterations of the associated loops of a loop-associated directive. The **order** clause is part of the schedule specification for the purpose of determining its consistency with other schedules (see Section 4.4.5). If *ordering* is **concurrent**, the logical iterations of the associated loops may execute in any order, including concurrently. The specified schedule is reproducible if the **reproducible** modifier is present. If *order-modifier* is not **unconstrained**, the behavior is as if the **reproducible** modifier is present.

Restrictions

Restrictions to the **order** clause are as follows:

- The only constructs that may be encountered inside a region that corresponds to a construct with an **order** clause that specifies **concurrent** are the **loop** construct, the **parallel** construct, the **simd** construct, and combined constructs for which the first construct is a **parallel** construct.
- A region that corresponds to a construct with an **order** clause that specifies **concurrent** may not contain calls to procedures that contain OpenMP directives.
- A region that corresponds to a construct with an **order** clause that specifies **concurrent** may not contain OpenMP runtime API calls.
- If a threadprivate variable is referenced inside a region that corresponds to a construct with an **order** clause that specifies **concurrent**, the behavior is unspecified.
- At most one **order** clause may appear on a construct.

10.4 simd Construct

Name: simd **Category:** executable	**Association:** loop **Properties:** parallelism-generating, context-matching, simdizable

Separating Directives:
scan

Clauses:
aligned, **collapse**, **if**, **lastprivate**, **linear**, **nontemporal**, **order**, **private**, **reduction**, **safelen**, **simdlen**

Binding

A **simd** region binds to the current task region. The binding thread set of the **simd** region is the current team.

Semantics

The **simd** construct enables the execution of multiple iterations of the associated loops concurrently by using SIMD instructions. At the beginning of each logical iteration, the loop iteration variable or the variable declared by *range-decl* of each associated loop has the value that it would have if the set of the associated loops was executed sequentially. The number of iterations that are executed concurrently at any given time is implementation defined. Each concurrent iteration will be executed by a different SIMD lane. Each set of concurrent iterations is a SIMD chunk. Lexical forward dependences in the iterations of the original loop must be preserved within each SIMD chunk, unless an **order** clause that specifies **concurrent** is present.

When an **if** clause is present and evaluates to *false*, the preferred number of iterations to be executed concurrently is one, regardless of whether a **simdlen** clause is specified.

Restrictions

Restrictions to the **simd** construct are as follows:

- If both **simdlen** and **safelen** clauses are specified, the value of the **simdlen** *length* must be less than or equal to the value of the **safelen** *length*.
- Only simdizable constructs can be encountered during execution of a **simd** region.
- If an **order** clause that specifies **concurrent** appears on a **simd** directive, the **safelen** clause may not also appear.

C / C++

- The **simd** region cannot contain calls to the **longjmp** or **setjmp** functions.

C / C++

C++

- No exception can be raised in the **simd** region.
- The only random access iterator types that are allowed for the associated loops are pointer types.

C++

Cross References

- **aligned** clause, see Section 6.3.
- **if** clause, see Section 3.4.
- Canonical loop nest form, see Section 4.4.1.
- Data-sharing attribute clauses, see Section 5.4.
- **order** clause, see Section 10.3.
- **nontemporal** clause, see Section 10.4.1.
- **safelen** clause, see Section 10.4.2.
- **simdlen** clause, see Section 10.4.3.

10.4.1 `nontemporal` Clause

Name:	Properties:
`nontemporal`	unique, positive constant

Arguments:

Name	Type	Properties
length	Expression of type integer	default

Directives:

`simd`

Semantics

The **`nontemporal`** clause specifies that accesses to the storage locations to which the list items refer have low temporal locality across the iterations in which those storage locations are accessed.

Cross References

- **`simd`** construct, see Section 10.4

10.4.2 `safelen` Clause

Name:	Properties:
`safelen`	unique

Arguments:

Name	Type	Properties
length	Expression of type integer	positive constant

Directives:

`simd`

Semantics

The **`safelen`** clause specifies that no two concurrent iterations within a SIMD chunk can have a distance in the logical iteration space that is greater than or equal to the value given in the clause. The parameter of the **`safelen`** clause must be a constant positive

Cross References

- **`simd`** construct, see Section 10.4

10.4.3 `simdlen` Clause

Name:	Properties:
`simdlen`	unique

Arguments:

Name	Type	Properties
length	Expression of type integer	positive constant

Directives:
declare simd, **simd**

Semantics
When the **simdlen** clause appears on a **simd** construct, *length* is treated as a hint that specifies the preferred number of iterations to be executed concurrently. When the **simdlen** clause appears on a **declare simd** construct, if a SIMD version of the associated function is created, *length* corresponds to the number of concurrent arguments of the function.

Cross References
- **declare simd** directive, see Section 7.7.
- **simd** construct, see Section 10.4

10.5 masked Construct

Name: **masked**	Association: block
Category: executable	**Properties:** thread-limiting

Clauses:
filter

Additional information: The **master** construct, which has been deprecated, has the same syntax as the **masked** construct other than the use of **master** as the directive name and that the **filter** clause may not be specified for the **master** construct.

Binding
The binding thread set for a **masked** region is the current team. A **masked** region binds to the innermost enclosing parallel region.

Semantics
The **masked** construct specifies a structured block that is executed by a subset of the threads of the current team. Only the threads of the team that executes the binding parallel region that the **filter** clause selects participate in the execution of the structured block of a **masked** region. Other threads in the team do not execute the associated structured block. No implied barrier occurs either on entry to or exit from the **masked** construct. The result of evaluating the *thread_num* parameter of the **filter** clause may vary across threads.

If more than one thread in the team executes the structured block of a **masked** region, the structured block must include any synchronization required to ensure that data races do not occur.

The **master** construct, which has been deprecated, has identical semantics to the **masked** construct with no **filter** clause present.

Execution Model Events

The *masked-begin* event occurs in any thread of a team that executes the **masked** region on entry to the region.

The *masked-end* event occurs in any thread of a team that executes the **masked** region on exit from the region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_masked** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *masked-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_masked** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *masked-end* event in that thread. These callbacks occur in the context of the task executed by the current thread and have the type signature **ompt_callback_masked_t**.

Cross References

- **ompt_callback_masked_t**, see Section 19.5.2.12.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 19.4.4.11.
- **parallel** construct, see Section 10.1.

10.5.1 filter Clause

Name:	**Properties:**
filter	unique

Arguments:

Name	Type	Properties
thread_num	Expression of type integer	default

Directives:

masked

Semantics

If *thread_num* specifies the thread number of the current thread in the current team then the **filter** clause selects the current thread. If the **filter** clause is not specified, the effect is as if *thread_num* evaluates to zero, so that the **filter** clause selects the primary thread. The use of a variable in a *thread_num* clause expression causes an implicit reference to the variable in all enclosing constructs.

Cross References

- **masked** construct, see Section 10.5

11 Work-Distribution Constructs

A work-distribution construct distributes the execution of the corresponding region among the threads in its binding thread set. Threads execute portions of the region in the context of the implicit tasks that each one is executing. A work-distribution construct is *worksharing* if the binding thread set is a thread team.

A worksharing region has no barrier on entry; however, an implied barrier exists at the end of the worksharing region, unless a **nowait** clause is specified. If a **nowait** clause is present, an implementation may omit the barrier at the end of the worksharing region. In this case, threads that finish early may proceed straight to the instructions that follow the worksharing region without waiting for the other members of the team to finish the worksharing region, and without performing a flush operation.

Restrictions

The following restrictions apply to work-distribution constructs:

- Each work-distribution region must be encountered by all threads in the binding thread set or by none at all, unless it is a worksharing region and cancellation has been requested for the innermost enclosing parallel region.
- The sequence of encountered work-distribution regions that have the same binding thread set must be the same for every thread in the binding thread set.
- The sequence of encountered worksharing regions and **barrier** regions that bind to the same thread team must be the same for every thread in the team.

11.1 single Construct

Name: single **Category:** executable	**Association:** block **Properties:** work-distribution, worksharing, thread-limiting

Clauses:
allocate, **copyprivate**, **firstprivate**, **nowait**, **private**

Binding

The binding thread set for a **single** region is the current team. A **single** region binds to the innermost enclosing **parallel** region. Only the threads of the team that executes the binding **parallel** region participate in the execution of the structured block and the implied barrier of the **single** region if the barrier is not eliminated by a **nowait** clause.

Semantics

The **single** construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the primary thread), in the context of its implicit task. The method of choosing a thread to execute the structured block each time the team encounters the construct is implementation defined. An implicit barrier occurs at the end of a **single** region if the **nowait** clause is not specified.

Execution Model Events

The *single-begin* event occurs after an implicit task encounters a **single** construct but before the task starts to execute the structured block of the **single** region.

The *single-end* event occurs after an implicit task finishes execution of a **single** region but before it resumes execution of the enclosing region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *single-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *single-end* event in that thread. For each of these callbacks, the *wstype* argument is **ompt_work_single_executor** if the thread executes the structured block associated with the **single** region; otherwise, the *wstype* argument is **ompt_work_single_other**. The callback has type signature **ompt_callback_work_t**.

Restrictions

Restrictions to the **single** construct are as follows:

- The **copyprivate** clause must not be used with the **nowait** clause.

Cross References

- **ompt_callback_work_t**, see Section 19.5.2.5.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 19.4.4.11.
- **ompt_work_single_executor** and **ompt_work_single_other**, see Section 19.4.4.16.
- **private** and **firstprivate** clauses, see Section 5.4.
- **allocate** clause, see Section 6.7.
- **nowait** clause, see Section 15.6.
- **copyprivate** clause, see Section 5.7.2.

11.2 scope Construct

Name: **scope** **Category:** executable	**Association:** block **Properties:** work-distribution, worksharing, thread-limiting

Clauses:
allocate, **firstprivate**, **nowait**, **private**, **reduction**

Binding

The binding thread set for a **scope** region is the current team. A **scope** region binds to the innermost enclosing parallel region. Only the threads of the team that executes the binding parallel region participate in the execution of the structured block and the implied barrier of the **scope** region if the barrier is not eliminated by a **nowait** clause.

Semantics

The **scope** construct specifies that all threads in a team execute the associated structured block and any additionally specified OpenMP operations. An implicit barrier occurs at the end of a **scope** region if the **nowait** clause is not specified.

Execution Model Events

The *scope-begin* event occurs after an implicit task encounters a **scope** construct but before the task starts to execute the structured block of the **scope** region.

The *scope-end* event occurs after an implicit task finishes execution of a **scope** region but before it resumes execution of the enclosing region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_work_scope** as its *work_type* argument for each occurrence of a *scope-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_work_scope** as its *work_type* argument for each occurrence of a *scope-end* event in that thread. The callbacks occur in the context of the implicit task. The callbacks have type signature **ompt_callback_work_t**.

Cross References

- **ompt_callback_work_t**, see Section 19.5.2.5.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 19.4.4.11.
- **ompt_work_scope**, see Section 19.4.4.16.
- **nowait** clause, see Section 15.6.
- **private** clause, Section 5.4.3.
- **reduction** clause, Section 5.5.9.

11.3 `sections` Construct

Name: **sections** Category: executable	Association: block Properties: work-distribution, worksharing, thread-limiting, cancellable

Separating Directives:
section

Clauses:
allocate, **firstprivate**, **lastprivate**, **nowait**, **private**, **reduction**

Binding

The binding thread set for a **sections** region is the current team. A **sections** region binds to the innermost enclosing **parallel** region. Only the threads of the team that executes the binding **parallel** region participate in the execution of the structured block sequences and the implied barrier of the **sections** region if the barrier is not eliminated by a **nowait** clause.

Semantics

The **sections** construct is a non-iterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team. Each structured block is executed once by one of the threads in the team in the context of its implicit task. An implicit barrier occurs at the end of a **sections** region if the **nowait** clause is not specified.

Each structured block sequence in the **sections** construct is preceded by a **section** directive except possibly the first sequence, for which a preceding **section** directive is optional. The method of scheduling the structured block sequences among the threads in the team is implementation defined.

Execution Model Events

The *sections-begin* event occurs after an implicit task encounters a **sections** construct but before the task executes any structured block sequences of the **sections** region.

The *sections-end* event occurs after an implicit task finishes execution of a **sections** region but before it resumes execution of the enclosing context.

The *section-begin* event occurs before an implicit task starts to execute a structured block sequence in the **sections** construct for each of those structured block sequences that the task executes.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_work_sections** as its *work_type* argument for each occurrence of a *sections-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_work_sections** as its *work_type* argument for each occurrence of a *sections-end* event in that thread. The callbacks occur in the context of the implicit task. The callbacks have type signature **ompt_callback_work_t**.

A thread dispatches a registered **ompt_callback_dispatch** callback for each occurrence of a *section-begin* event in that thread. The callback occurs in the context of the implicit task. The callback has type signature **ompt_callback_dispatch_t**.

Cross References

- **ompt_callback_dispatch_t**, see Section 19.5.2.6.
- **ompt_callback_work_t**, see Section 19.5.2.5.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 19.4.4.11.
- **ompt_work_sections**, see Section 19.4.4.16.
- **private**, **firstprivate**, **lastprivate**, and **reduction** clauses, see Section 5.4.
- **allocate** clause, see Section 6.7.
- **nowait** clause, see Section 15.6.
- **section** directive, see Section 11.3.1.

11.3.1 section Directive

Name: section **Category:** subsidiary	**Association:** separating **Properties:** default

Separated Directives:
sections

Semantics

The **section** directive may be used to separate the structured block that is associated with a **sections** construct into multiple sections.

Cross References

- **sections** construct, see Section 11.3.

11.4 `workshare` Construct

Name: **`workshare`**	**Association:** block
Category: executable	**Properties:** work-distribution, worksharing

Clauses:
nowait

Binding

The binding thread set for a **workshare** region is the current team. A **workshare** region binds to the innermost enclosing **parallel** region. Only the threads of the team that executes the binding **parallel** region participate in the execution of the units of work and the implied barrier of the **workshare** region if the barrier is not eliminated by a **nowait** clause.

Semantics

The **workshare** construct divides the execution of the associated structured block into separate units of work and causes the threads of the team to share the work such that each unit is executed only once by one thread, in the context of its implicit task. An implicit barrier occurs at the end of a **workshare** region if a **nowait** clause is not specified.

An implementation of the **workshare** construct must insert any synchronization that is required to maintain standard Fortran semantics. For example, the effects of one statement within the structured block must appear to occur before the execution of succeeding statements, and the evaluation of the right hand side of an assignment must appear to complete prior to the effects of assigning to the left hand side.

The statements in the **workshare** construct are divided into units of work as follows:

- For array expressions within each statement, including transformational array intrinsic functions that compute scalar values from arrays:
  - Evaluation of each element of the array expression, including any references to elemental functions, is a unit of work.
  - Evaluation of transformational array intrinsic functions may be freely subdivided into any number of units of work.
- For array assignment statements, assignment of each element is a unit of work.
- For scalar assignment statements, each assignment operation is a unit of work.
- For **WHERE** statements or constructs, evaluation of the mask expression and the masked assignments are each a unit of work.
- For **FORALL** statements or constructs, evaluation of the mask expression, expressions occurring in the specification of the iteration space, and the masked assignments are each a unit of work.

- For **atomic** constructs, **critical** constructs, and **parallel** constructs, the construct is a unit of work. A new thread team executes the statements contained in a **parallel** construct.
- If none of the rules above apply to a portion of a statement in the structured block, then that portion is a unit of work.

The transformational array intrinsic functions are **MATMUL**, **DOT_PRODUCT**, **SUM**, **PRODUCT**, **MAXVAL**, **MINVAL**, **COUNT**, **ANY**, **ALL**, **SPREAD**, **PACK**, **UNPACK**, **RESHAPE**, **TRANSPOSE**, **EOSHIFT**, **CSHIFT**, **MINLOC**, and **MAXLOC**.

How units of work are assigned to the threads that execute a **workshare** regionis unspecified.

If an array expression in the block references the value, association status, or allocation status of private variables, the value of the expression is undefined, unless the same value would be computed by every thread.

If an array assignment, a scalar assignment, a masked array assignment, or a **FORALL** assignment assigns to a private variable in the block, the result is unspecified.

The **workshare** directive causes the sharing of work to occur only in the **workshare** construct, and not in the remainder of the **workshare** region.

Execution Model Events

The *workshare-begin* event occurs after an implicit task encounters a **workshare** construct but before the task starts to execute the structured block of the **workshare** region.

The *workshare-end* event occurs after an implicit task finishes execution of a **workshare** region but before it resumes execution of the enclosing context.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_work_workshare** as its *work_type* argument for each occurrence of a *workshare-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_work_workshare** as its *work_type* argument for each occurrence of a *workshare-end* event in that thread. The callbacks occur in the context of the implicit task. The callbacks have type signature **ompt_callback_work_t**.

Restrictions

Restrictions to the **workshare** construct are as follows:

- The only OpenMP constructs that may be closely nested inside a **workshare** construct are the **atomic**, **critical**, and **parallel** constructs.
- Base language statements that are encountered inside a **workshare** construct but that are not enclosed within a **parallel** construct that is nested inside the **workshare** construct must consist of only the following:

- array assignments;
- scalar assignments;
- **FORALL** statements;
- **FORALL** constructs;
- **WHERE** statements; and
- **WHERE** constructs.

- All array assignments, scalar assignments, and masked array assignments that are encountered inside a **workshare** construct but are not nested inside a **parallel** construct that is nested inside the **workshare** construct must be intrinsic assignments.
- The construct must not contain any user-defined function calls unless either the function is pure and elemental or the function call is contained inside a **parallel** construct that is nested inside the **workshare** construct.

Cross References

- **ompt_callback_work_t**, see Section 19.5.2.5.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 19.4.4.11.
- **ompt_work_workshare**, see Section 19.4.4.16.
- **parallel** construct, see Section 10.1.
- **atomic** construct, see Section 15.8.4.
- **critical** construct, see Section 15.2.
- **nowait** clause, see Section 15.6.

Fortran

11.5 Worksharing-Loop Constructs

Binding

The binding thread set for a worksharing-loop region is the current team. A worksharing-loop region binds to the innermost enclosing **parallel** region.

Semantics

The worksharing-loop construct is a worksharing construct that specifies that the iterations of one or more associated loops will be executed in parallel by threads in the team in the context of their implicit tasks. The iterations are distributed across threads that already exist in the team that is executing the **parallel** region to which the worksharing-loop region binds. Only those threads participate in execution of the loop iterations and the implied barrier of the worksharing-loop region when that barrier is not eliminated by a **nowait** clause. Each thread executes its assigned chunks in the context of its implicit task. The iterations of a given chunk are executed in sequential order.

If specified, the **schedule** clause determines the schedule of the logical iterations associated with the construct. That is, it determines the division of iterations into chunks and how those chunks are assigned to the threads. If the **schedule** clause is not specified then the schedule is implementation defined.

At the beginning of each logical iteration, the loop iteration variable or the variable declared by *range-decl* of each associated loop has the value that it would have if the set of the associated loops was executed sequentially.

The schedule is reproducible if one of the following conditions is true:

- The **order** clause is specified with the **reproducible** *order-modifier*; or
- The **schedule** clause is specified with **static** as the *kind* argument but not the **simd** *ordering-modifier*.

Programs can only depend on which thread executes a particular iteration if the schedule is reproducible. Schedule reproducibility also determines its consistency with other schedules.

Execution Model Events

The *ws-loop-begin* event occurs after an implicit task encounters a worksharing-loop construct but before the task starts execution of the structured block of the worksharing-loop region.

The *ws-loop-end* event occurs after a worksharing-loop region finishes execution but before resuming execution of the encountering task.

The *ws-loop-iteration-begin* event occurs at the beginning of each iteration of a worksharing-loop region. The *ws-loop-chunk-begin* event occurs for each scheduled chunk of a worksharing-loop region before the implicit task executes any of the associated iterations.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *ws-loop-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *ws-loop-end* event in that thread. The callbacks occur in the context of the implicit task. The callbacks have type signature **ompt_callback_work_t** and the *work_type* argument indicates the schedule as shown in Table 11.1.

A thread dispatches a registered **ompt_callback_dispatch** callback for each occurrence of a *ws-loop-iteration-begin* or *ws-loop-chunk-begin* event in that thread. The callback occurs in the context of the implicit task. The callback has type signature **ompt_callback_dispatch_t**.

TABLE 11.1: ompt_callback_work Callback Work Types for Worksharing-Loop

Value of *work_type*	If determined schedule is
ompt_work_loop	unknown at runtime
ompt_work_loop_static	**static**
ompt_work_loop_dynamic	**dynamic**
ompt_work_loop_guided	**guided**
ompt_work_loop_other	implementation specific

Restrictions

Restrictions to the worksharing-loop construct are as follows:

- The values of the loop control expressions of the loops associated with the worksharing-loop construct must be the same for all threads in the team.
- The value of the *run-sched-var* ICV must be the same for all threads in the team.

Cross References

- **OMP_SCHEDULE** environment variable, see Section 21.2.1.
- **ompt_callback_work_t**, see Section 19.5.2.5.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 19.4.4.11.
- **ompt_work_loop**, see Section 19.4.4.16.
- Consistent loop schedules, see Section 4.4.5).
- **order** clause, see Section 10.3.
- **do** construct, see Section 11.5.2.
- **for** construct, see Section 11.5.1.

- **nowait** clause, see Section 15.6.
- **schedule** clause, see Section 11.5.3.

C / C++

11.5.1 for Construct

Name: **for** **Category:** executable	**Association:** loop-associated **Properties:** work-distribution, worksharing, worksharing-loop, cancellable, context-matching

Separating Directives:
scan

Clauses:
allocate, **collapse**, **firstprivate**, **lastprivate**, **linear**, **nowait**, **order**, **ordered**, **private**, **reduction**, **schedule**

Semantics

The **for** is a worksharing-loop construct.

Cross References

- Worksharing-loop constructs, see Section 11.5.
- Canonical loop nest form, see Section 4.4.1.
- Data-sharing attribute clauses, see Section 5.4.
- **order** clause, see Section 10.3.
- **nowait** clause, see Section 15.6.
- **ordered** construct, see Section 15.9.7.

C / C++

Fortran

11.5.2 do Construct

Name: **do** **Category:** executable	**Association:** loop **Properties:** work-distribution, worksharing, worksharing-loop, cancellable, context-matching

Separating Directives:
scan

Clauses:
allocate, **collapse**, **firstprivate**, **lastprivate**, **linear**, **nowait**, **order**, **ordered**, **private**, **reduction**, **schedule**

Semantics

The **do** is a worksharing-loop construct.

Cross References

- Worksharing-loop constructs, see Section 11.5.
- Canonical loop nest form, see Section 4.4.1.
- Data-sharing attribute clauses, see Section 5.4.
- **order** clause, see Section 10.3.
- **nowait** clause, see Section 15.6.
- **ordered** construct, see Section 15.9.7.

Fortran

11.5.3 schedule Clause

Name:	Properties:
schedule	unique

Arguments:

Name	Type	Properties
kind	**Keyword:** **auto**, **dynamic**, **guided**, **runtime**, **static**	default
chunk_size	Expression of type integer	ultimate, optional, positive, region-invariant

Modifiers:

Name	Modifies	Type	Properties
ordering-modifier	*kind*	**Keyword:** **`monotonic`**, **`nonmonotonic`**	unique
chunk-modifier	*kind*	**Keyword:** **`simd`**	unique

Directives:

`do`, **`for`**

Semantics

The **`schedule`** clause specifies how iterations of associated loops of a woksharing-loop construct are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team. The *chunk_size* expression is evaluated using the original list items of any variables that are made private in the worksharing-loop construct. Whether, in what order, or how many times, any side effects of the evaluation of this expression occur is unspecified. The use of a variable in a **`schedule`** clause expression of a worksharing-loop construct causes an implicit reference to the variable in all enclosing constructs.

If the *kind* argument is **`static`**, iterations are divided into chunks of size *chunk_size*, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number. Each chunk contains *chunk_size* iterations, except for the chunk that contains the sequentially last iteration, which may have fewer iterations. If *chunk_size* is not specified, the logical iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. The size of the chunks is unspecified in this case.

If the *kind* argument is **`dynamic`**, the iterations are distributed to threads in the team in chunks. Each thread executes a chunk, then requests another chunk, until no chunks remain to be distributed. Each chunk contains *chunk_size* iterations, except for the chunk that contains the sequentially last iteration, which may have fewer iterations. If *chunk_size* is not specified, it defaults to 1.

If the *kind* argument is **`guided`**, the iterations are assigned to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a *chunk_size* of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1. For a *chunk_size* with value $k > 1$, the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than k iterations (except for the chunk that contains the sequentially last iteration, which may have fewer than k iterations). If *chunk_size* is not specified, it defaults to 1.

If the *kind* argument is **`auto`**, the decision regarding scheduling is delegated to the compiler and/or runtime system. The programmer gives the implementation the freedom to choose any possible mapping of iterations to threads in the team.

If the *kind* argument is **`runtime`**, the decision regarding scheduling is deferred until run time, and the schedule and chunk size are taken from the *run-sched-var* ICV. If the ICV is set to **`auto`**, the schedule is implementation defined. If the **`schedule`** clause specifies any modifiers then they override any corresponding modifiers that are specified in the *run-sched-var* ICV.

If the **simd** *chunk-modifier* is specified and the loop is associated with a SIMD construct, $new\_chunk\_size = \lceil chunk\_size/simd\_width \rceil * simd\_width$ is the *chunk_size* for all chunks except the first and last chunks, where *simd_width* is an implementation-defined value. The first chunk will have at least *new_chunk_size* iterations except if it is also the last chunk. The last chunk may have fewer iterations than *new_chunk_size*. If the **simd** modifier is specified and the loop is not associated with a SIMD construct, the modifier is ignored.

Note – For a team of p threads and a loop of n iterations, let $\lceil n/p \rceil$ be the integer q that satisfies $n = p * q - r$, with $0 <= r < p$. One compliant implementation of the **static** schedule (with no specified *chunk_size*) would behave as though *chunk_size* had been specified with value q. Another compliant implementation would assign q iterations to the first $p - r$ threads, and $q - 1$ iterations to the remaining r threads. This illustrates why a conforming program must not rely on the details of a particular implementation.

A compliant implementation of the **guided** schedule with a *chunk_size* value of k would assign $q = \lceil n/p \rceil$ iterations to the first available thread and set n to the larger of $n - q$ and $p * k$. It would then repeat this process until q is greater than or equal to the number of remaining iterations, at which time the remaining iterations form the final chunk. Another compliant implementation could use the same method, except with $q = \lceil n/(2p) \rceil$, and set n to the larger of $n - q$ and $2 * p * k$.

If the **monotonic** *ordering-modifier* is specified then each thread executes the chunks that it is assigned in increasing logical iteration order. When the **nonmonotonic** *ordering-modifier* is specified then chunks may be assigned to threads in any order and the behavior of an application that depends on any execution order of the chunks is unspecified. If an *ordering-modifier* is not specified, the effect is as if the **monotonic** modifier is specified if the *kind* argument is **static** or an **ordered** clause is specified on the construct; otherwise, the effect is as if the **nonmonotonic** modifier is specified.

Restrictions

Restrictions to the **schedule** clause are as follows:

- The **schedule** clause cannot be specified if any of the associated loops are non-rectangular.
- The value of the *chunk_size* expression must be the same for all threads in the team.
- If **runtime** or **auto** is specified for *kind*, *chunk_size* must not be specified.
- The **nonmonotonic** *ordering-modifier* cannot be specified if an **ordered** clause is specified on the same construct.

Cross References

- ICVs, see Section 2.
- Worksharing-loop constructs, see Section 11.5.

- **do** construct, see Section 11.5.2
- **for** construct, see Section 11.5.1
- **ordered** clause, see Section 4.4.4

11.6 distribute Construct

Name: distribute	**Association:** loop
Category: executable	**Properties:** work-distribution

Clauses:
allocate, **collapse**, **dist_schedule**, **firstprivate**, **lastprivate**, **order**, **private**

Binding

The binding thread set for a **distribute** region is the set of initial threads executing an enclosing **teams** region. A **distribute** region binds to this **teams** region.

Semantics

The **distribute** construct specifies that the iterations of one or more loops will be executed by the initial teams in the context of their implicit tasks. The iterations are distributed across the initial threads of all initial teams that execute the **teams** region to which the **distribute** region binds. No implicit barrier occurs at the end of a **distribute** region. To avoid data races the original list items that are modified due to **lastprivate** or **linear** clauses should not be accessed between the end of the **distribute** construct and the end of the **teams** region to which the **distribute** binds.

If the **dist_schedule** clause is not specified, the schedule is implementation defined.

At the beginning of each logical iteration, the loop iteration variable or the variable declared by *range-decl* of each associated loop has the value that it would have if the set of the associated loops was executed sequentially.

The schedule is reproducible if one of the following conditions is true:

- The **order** clause is present and uses the **reproducible** modifier; or
- The **dist_schedule** clause is specified with **static** as the *kind* parameter.

Programs can only depend on which team executes a particular iteration if the schedule is reproducible. Schedule reproducibility is also used for determining its consistency with other schedules.

Execution Model Events

The *distribute-begin* event occurs after an initial task encounters a **distribute** construct but before the task starts to execute the structured block of the **distribute** region.

The *distribute-end* event occurs after an initial task finishes execution of a **distribute** region but before it resumes execution of the enclosing context.

The *distribute-chunk-begin* event occurs for each scheduled chunk of a **distribute** region before execution of any associated iteration.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_work_distribute** as its *work_type* argument for each occurrence of a *distribute-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_work_distribute** as its *work_type* argument for each occurrence of a *distribute-end* event in that thread. The callbacks occur in the context of the implicit task. The callbacks have type signature **ompt_callback_work_t**.

A thread dispatches a registered **ompt_callback_dispatch** callback for each occurrence of a *distribute-chunk-begin* event in that thread. The callback occurs in the context of the initial task. The callback has type signature **ompt_callback_dispatch_t**.

Restrictions

Restrictions to the **distribute** construct are as follows:

- The values of the loop control expressions of the loops associated with the **distribute** construct must be the same for all teams in the league.
- The region that corresponds to the **distribute** construct must be strictly nested inside a **teams** region.
- A list item may appear in a **firstprivate** or **lastprivate** clause, but not in both.
- The **conditional** *lastprivate-modifier* must not be specified.

Cross References

- **ompt_callback_work_t**, see Section 19.5.2.5.
- **ompt_work_distribute**, see Section 19.4.4.16.
- **teams** construct, see Section 10.2
- Canonical loop nest form, see Section 4.4.1.
- Consistent loop schedules, see Section 4.4.5).
- **order** clause, see Section 10.3.
- **dist_schedule** clause, see Section 11.6.1.

11.6.1 `dist_schedule` Clause

Name: **`dist_schedule`**

Properties: unique

Arguments:

Name	Type	Properties
kind	**Keyword: `static`**	default
chunk_size	Expression of type integer	ultimate, optional, positive, region-invariant

Directives:

`distribute`

Semantics

The **`dist_schedule`** clause specifies how iterations of associated loops of a **`distribute`** construct are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among the teams of the league. if *chunk_size* is not specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each initial team of the league.

If the *chunk_size* argument is specified, iterations are divided into chunks of size *chunk_size*. The *chunk_size* expression is evaluated using the original list items of any variables that are made private in the **`distribute`** construct. Whether, in what order, or how many times, any side effects of the evaluation of this expression occur is unspecified. The use of a variable in a **`dist_schedule`** clause expression of a **`distribute`** construct causes an implicit reference to the variable in all enclosing constructs. These chunks are assigned to the initial teams of the league in a round-robin fashion in the order of the initial team number.

Restrictions

Restrictions to the **`dist_schedule`** clause are as follows:

- The value of the *chunk_size* expression must be the same for all teams in the league.
- The **`dist_schedule`** clause cannot be specified if any of the associated loops are non-rectangular.

Cross References

- **`distribute`** construct, see Section 11.6

11.7 `loop` Construct

Name: `loop` Category: executable	Association: loop-associated Properties: work-distribution, worksharing, simdizable

Clauses:
bind, **collapse**, **lastprivate**, **order**, **private**, **reduction**

Binding

The **bind** clause determines the binding region, which determines the binding thread set.

Semantics

A **loop** construct specifies that the logical iterations of the associated loops may execute concurrently and permits the encountering threads to execute the loop accordingly. A **loop** construct is a worksharing construct if its binding region is the innermost enclosing parallel region. Otherwise it is not a worksharing region. The directive asserts that the iterations of the associated loops may execute in any order, including concurrently. Each logical iteration is executed once per instance of the **loop** region that is encountered by exactly one thread that is a member of the binding thread set.

At the beginning of each logical iteration, the loop iteration variable or the variable declared by *range-decl* of each associated loop has the value that it would have if the set of the associated loops was executed sequentially.

If the **order** clause is not present, the behavior is as if an **order** clause that specifies **concurrent** appeared on the construct.

If the **loop** region binds to a **teams** region, the threads in the binding thread set may continue execution after the **loop** region without waiting for all logical iterations of the associated loops to complete. The iterations are guaranteed to complete before the end of the **teams** region. If the **loop** region does not bind to a **teams** region, all logical iterations of the associated loops must complete before the encountering threads continue execution after the **loop** region.

For the purpose of determining its consistency with other schedules, the schedule is defined by the implicit **order** clause. The schedule is reproducible if the schedule specified through the implicit **order** clause is reproducible.

Restrictions

Restrictions to the **loop** construct are as follows:

- A list item may not appear in a **lastprivate** clause unless it is the loop iteration variable of a loop that is associated with the construct.
- If a *reduction-modifier* is specified in a **reduction** clause that appears on the directive then the reduction modifer must be **default**.

- If a **loop** construct is not nested inside another OpenMP construct and it appears in a procedure, the **bind** clause must be present.
- If a **loop** region binds to a **teams** or parallel region, it must be encountered by all threads in the binding thread set or by none of them.

Cross References

- Worksharing-Loop construct, see Section 11.5.
- **simd** construct, see Section 10.4.
- Canonical loop nest form, see Section 4.4.1.
- Consistent loop schedules, see Section 4.4.5).
- **order** clause, see Section 10.3.
- **bind** clause, see Section 11.7.1.
- **distribute** construct, see Section 11.6.
- **single** construct, see Section 11.1.

11.7.1 bind Clause

Name:	**Properties:**
bind	unique

Arguments:

Name	Type	Properties
binding	**Keyword: parallel, teams, thread**	default

Directives:
loop

Semantics

The **bind** clause specifies the binding region of the construct on which it appears. Specifically, if *binding* is **teams** and an innermost enclosing **teams** region exists then the binding region is that **teams** region; if *binding* is **parallel** then the binding region is the innermost enclosing parallel region, which may be an implicit parallel region; and if *binding* is **thread** then the binding region is not defined. If the **bind** clause is not specified on a construct for which it may be specified and the construct is closely nested inside a **teams** or **parallel** construct, the effect is as if *binding* is **teams** or **parallel**. If none of those conditions hold, the binding region is not defined.

The specified binding region determines the binding thread set. Specifically, if the binding region is a **teams** region, then the binding thread set is the set of initial threads that are executing that region while if the binding region is a parallel region, then the binding thread set is the team of threads that are executing that region. If the binding region is not defined, then the binding thread set is the encountering thread.

Restrictions

Restrictions to the **bind** clause are as follows:

- If **teams** is specified as *binding* then the corresponding **loop** region must be strictly nested inside a **teams** region.
- If **teams** is specified as *binding* and the corresponding **loop** region executes on a non-host device then the behavior of a **reduction** clause that appears on the corresponding **loop** construct is unspecified if the construct is not nested inside a **teams** construct.
- If **parallel** is specified as *binding*, the behavior is unspecified if the corresponding **loop** region is closely nested inside a **simd** region.

Cross References

- **loop** construct, see Section 11.7

12 Tasking Constructs

This chapter defines directives and concepts related to explicit tasks.

12.1 untied Clause

Name:	Properties:
untied	unique, inarguable

Directives:
task, **taskloop**

Semantics

The **untied** clause specifies that tasks generated by the construct on which it appears are untied, which means that any thread in the team can resume the **task** region after a suspension. If the **untied** clause is not specified on a construct on which it may appear, generated tasks are tied; if a tied task is suspended, its **task** region can only be resumed by the thread that started its execution. If a generated task is a final or an included task, the **untied** clause is ignored and the task is tied.

Cross References

- **task** construct, see Section 12.5.
- **taskloop** construct, see Section 12.6.

12.2 mergeable Clause

Name:	Properties:
mergeable	unique, inarguable

Directives:
task, **taskloop**

Semantics

The **mergeable** clause specifies that tasks generated by the construct on which it appears are mergeable tasks.

Cross References

- **`task`** construct, see Section 12.5.
- **`taskloop`** construct, see Section 12.6.

12.3 `final` Clause

Name: **`final`**

Properties: unique

Arguments:

Name	Type	Properties
finalize	Expression of type logical	default

Directives:

`task`, **`taskloop`**

Semantics

The **`final`** clause specifies that tasks generated by the construct on which it appears are final tasks if the *finalize* expression evaluates to *true*, All **`task`** constructs that are encountered during execution of a final task generate final and included tasks. The use of a variable in a *finalize* expression causes an implicit reference to the variable in all enclosing constructs. The *finalize* expression is evaluated in the context outside of the construct on which the clause appears,

Cross References

- **`task`** construct, see Section 12.5.
- **`taskloop`** construct, see Section 12.6.

12.4 `priority` Clause

Name: **`priority`**

Properties: unique

Arguments:

Name	Type	Properties
priority-value	Expression of type integer	constant, non-negative

Directives:

`task`, **`taskloop`**

Semantics

The **`priority`** clause specifies a hint for the task execution order of tasks generated by the construct on which it appears in the *priority-value* argument. Among all tasks ready to be executed, higher priority tasks (those with a higher numerical *priority-value*) are recommended to execute before lower priority ones. The default *priority-value* when no **`priority`** clause is specified is zero (the lowest priority). If a specified *priority-value* is higher than the *max-task-priority-var* ICV then the implementation will use the value of that ICV. A program that relies on the task execution order being determined by the *priority-value* may have unspecified behavior.

Cross References

- **`task`** construct, see Section 12.5.
- **`taskloop`** construct, see Section 12.6.

12.5 `task` Construct

Name: **`task`** **Category:** executable	**Association:** block **Properties:** parallelism-generating, thread-limiting, task-generating

Clauses:

`affinity`, **`allocate`**, **`default`**, **`detach`**, **`final`**, **`firstprivate`**, **`if`**, **`in_reduction`**, **`mergeable`**, **`priority`**, **`private`**, **`shared`**, **`untied`**

Clause set:

Properties: fully exclusive	**Members:** mergeable, detach

Binding

The binding thread set of the **`task`** region is the current team. A **`task`** region binds to the innermost enclosing **`parallel`** region.

Semantics

When a thread encounters a **`task`** construct, an explicit task is generated from the code for the associated structured block. The data environment of the task is created according to the data-sharing attribute clauses on the **`task`** construct, per-data environment ICVs, and any defaults that apply. The data environment of the task is destroyed when the execution code of the associated structured block is completed.

The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task. Completion of the task can be guaranteed using task synchronization constructs and clauses. If a **`task`** construct is encountered during execution of an outer task, the generated **`task`** region that corresponds to this construct is not a part of the outer task region unless the generated task is an included task.

A detachable task is completed when the execution of its associated structured block is completed and the *allow-completion* event is fulfilled. If no **detach** clause is present on a **task** construct, the generated task is completed when the execution of its associated structured block is completed.

A thread that encounters a task scheduling point within the **task** region may temporarily suspend the **task** region.

The **task** construct includes a task scheduling point in the task region of its generating task, immediately following the generation of the explicit task. Each explicit **task** region includes a task scheduling point at the end of its associated structured block.

Note – When storage is shared by an explicit **task** region, the programmer must ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the explicit **task** region completes its execution.

When an **if** clause is present on a **task** construct and the **if** clause expression evaluates to *false*, an undeferred task is generated, and the encountering thread must suspend the current task region, for which execution cannot be resumed until execution of the structured block that is associated with the generated task is completed. The use of a variable in an **if** clause expression of a **task** construct causes an implicit reference to the variable in all enclosing constructs. The **if** clause expression is evaluated in the context outside of the **task** construct.

Execution Model Events

The *task-create* event occurs when a thread encounters a construct that causes a new task to be created. The event occurs after the task is initialized but before it begins execution or is deferred.

Tool Callbacks

A thread dispatches a registered **ompt_callback_task_create** callback for each occurrence of a *task-create* event in the context of the encountering task. This callback has the type signature **ompt_callback_task_create_t** and the *flags* argument indicates the task types shown in Table 12.1.

TABLE 12.1: `ompt_callback_task_create` Callback Flags Evaluation

Operation	Evaluates to true
(*flags* & **ompt_task_explicit**)	Always in the dispatched callback
(*flags* & **ompt_task_undeferred**)	If the task is an undeferred task
(*flags* & **ompt_task_final**)	If the task is a final task
(*flags* & **ompt_task_untied**)	If the task is an untied task
(*flags* & **ompt_task_mergeable**)	If the task is a mergeable task
(*flags* & **ompt_task_merged**)	If the task is a merged task

Cross References

- **final** clause, see Section 12.3.
- **if** clause, see Section 3.4.
- **mergeable** clause, see Section 12.2.
- **ompt_callback_task_create_t**, see Section 19.5.2.7.
- **priority** clause, see Section 12.4.
- **untied** clause, see Section 12.1.
- Data-sharing attribute clauses, see Section 5.4.
- Task scheduling constraints, see Section 12.9.
- **affinity** clause, see Section 12.5.1.
- **allocate** clause, see Section 6.7.
- **depend** clause, see Section 15.9.5.
- **detach** clause, see Section 12.5.2.
- **omp_fulfill_event**, see Section 18.11.1.
- **in_reduction** clause, see Section 5.5.11.

12.5.1 affinity Clause

Name: **affinity**

Properties: unique

Arguments:

Name	Type	Properties
locator-list	List containing locator list item	default

Modifiers:

Name	Modifies	Type	Properties
aff-modifier	*locator-list*	iterator modifier	unique

Directives:

task

Semantics

The **affinity** clause specifies a hint to indicate data affinity of tasks generated by the construct on which it appears. The hint recommends to execute generated tasks close to the location of the list items. A program that relies on the task execution location being determined by this list may have unspecified behavior.

The list items that appear in the **affinity** clause may reference iterators defined by an *iterators-definition* that appears in the same clause. The list items that appear in the **affinity** clause may include array sections.

C / C++

The list items that appear in the **affinity** clause may use shape-operators.

C / C++

If a list item appears in an **affinity** clause then data affinity refers to the original list item.

Cross References

- **task** construct, see Section 12.5.

12.5.2 detach Clause

Name: **detach**

Properties: unique

Arguments:

Name	Type	Properties
event-handle	Variable of type event_handle	default

Directives:

task

Semantics

The **detach** clause specifies that the task generated by the construct on which it appears is a detachable task. A new *allow-completion* event is created and connected to the completion of the associated **task** region. The original *event-handle* is updated to represent that *allow-completion* event before the task data environment is created. The *event-handle* is considered as if it was specified on a **firstprivate** clause. The use of a variable in a **detach** clause expression of a **task** construct causes an implicit reference to the variable in all enclosing constructs.

Restrictions

Restrictions to the **detach** clause are as follows:

- If a **detach** clause appears on a directive, then the encountering task must not be a final task.
- A variable that appears in a **detach** clause cannot appear in a data-sharing attribute clause on the same construct.
- A variable that is part of another variable (as an array element or a structure element) cannot appear in a **detach** clause.

Fortran

- *event-handle* must not have the **POINTER** attribute.
- If *event-handle* has the **ALLOCATABLE** attribute, the allocation status must be allocated when the **task** construct is encountered, and the allocation status must not be changed, either explicitly or implicitly, in the **task** region.

Fortran

Cross References

- **task** construct, see Section 12.5.

12.6 taskloop Construct

Name: taskloop **Category:** executable	**Association:** loop **Properties:** parallelism-generating, task-generating

Clauses:
allocate, **collapse**, **default**, **final**, **firstprivate**, **grainsize**, **if**, **in_reduction**, **lastprivate**, **mergeable**, **nogroup**, **num_tasks**, **priority**, **private**, **reduction**, **shared**, **untied**

Clause set:

Properties: fully exclusive	**Members:** reduction, nogroup

Binding

The binding thread set of the **taskloop** region is the current team. A **taskloop** region binds to the innermost enclosing **parallel** region.

Semantics

When a thread encounters a **taskloop** construct, the construct partitions the iterations of the associated loops into chunks, each of which is assigned to an explicit task for parallel execution. The iteration count for each associated loop is computed before entry to the outermost loop. The data environment of each generated task is created according to the data-sharing attribute clauses on the **taskloop** construct, per-data environment ICVs, and any defaults that apply. The order of the creation of the loop tasks is unspecified. Programs that rely on any execution order of the logical iterations are non-conforming.

By default, the **taskloop** construct executes as if it was enclosed in a **taskgroup** construct with no statements or directives outside of the **taskloop** construct. Thus, the **taskloop** construct creates an implicit **taskgroup** region. If the **nogroup** clause is present, no implicit **taskgroup** region is created.

If a **reduction** clause is present, the behavior is as if a **task_reduction** clause with the same reduction operator and list items was applied to the implicit **taskgroup** construct that encloses the **taskloop** construct. The **taskloop** construct executes as if each generated task was defined by a **task** construct on which an **in_reduction** clause with the same reduction operator and list items is present. Thus, the generated tasks are participants of the reduction defined by the **task_reduction** clause that was applied to the implicit **taskgroup** construct.

If an **in_reduction** clause is present, the behavior is as if each generated task was defined by a **task** construct on which an **in_reduction** clause with the same reduction operator and list items is present. Thus, the generated tasks are participants of a reduction previously defined by a reduction scoping clause.

If neither a **grainsize** nor **num_tasks** clause is present, the number of loop tasks generated and the number of logical iterations assigned to these tasks is implementation defined.

At the beginning of each logical iteration, the loop iteration variable or the variable declared by *range-decl* of each associated loop has the value that it would have if the set of the associated loops was executed sequentially.

When an **if** clause is present and the **if** clause expression evaluates to *false*, undeferred tasks are generated. The use of a variable in an **if** clause expression causes an implicit reference to the variable in all enclosing constructs.

C++

For **firstprivate** variables of class type, the number of invocations of copy constructors that perform the initialization is implementation defined.

C++

Note – When storage is shared by a **taskloop** region, the programmer must ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the **taskloop** region and its descendant tasks complete their execution.

Execution Model Events

The *taskloop-begin* event occurs upon entering the **taskloop** region. A *taskloop-begin* will precede any *task-create* events for the generated tasks. The *taskloop-end* event occurs upon completion of the **taskloop** region.

Events for an implicit taskgroup region that surrounds the **taskloop** region are the same as for the **taskgroup** construct.

The *taskloop-iteration-begin* event occurs at the beginning of each iteration of a **taskloop** region before an explicit task executes the iteration. The *taskloop-chunk-begin* event occurs before an explicit task executes any of its associated iterations in a **taskloop** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback for each occurrence of a *taskloop-begin* and *taskloop-end* event in that thread. The callback occurs in the context of the encountering task. The callback has type signature **ompt_callback_work_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate, and **ompt_work_taskloop** as its *work_type* argument.

A thread dispatches a registered **ompt_callback_dispatch** callback for each occurrence of a *taskloop-iteration-begin* or *taskloop-chunk-begin* event in that thread.

The callback binds to the explicit task executing the interations. The callback has type signature **ompt_callback_dispatch_t**.

Restrictions

Restrictions to the **taskloop** construct are as follows:

- The values of the loop control expressions of the loops associated with the **taskloop** construct must be region invariant.
- The *reduction-modifier* must be **default**.
- The **conditional** *lastprivate-modifier* must not be specified.

Cross References

- **final** clause, see Section 12.3.
- **if** clause, see Section 3.4.
- **mergeable** clause, see Section 12.2.
- **nogroup** clause, Section 15.7.
- **ompt_callback_dispatch_t**, see Section 19.5.2.6.
- **ompt_callback_work_t**, see Section 19.5.2.5.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 19.4.4.11.
- **ompt_work_taskloop**, see Section 19.4.4.16.
- **priority** clause, see Section 12.4.
- **untied** clause, see Section 12.1.
- Canonical loop nest form, see Section 4.4.1.
- Data-sharing attribute clauses, see Section 5.4.
- Reduction clauses and directives, see Section 5.5.
- **grainsize** clause, see Section 12.6.1.
- **num_tasks** clause, see Section 12.6.2.
- **task** construct, Section 12.5.
- **taskgroup** construct, Section 15.4.
- **tile** construct, see Section 9.1.

12.6.1 grainsize Clause

Name:	**Properties:**
grainsize	unique

Arguments:

Name	Type	Properties
grain-size	Expression of type integer	positive

Modifiers:

Name	Modifies	Type	Properties
prescriptiveness	*grain-size*	**Keyword: strict**	unique

Directives:

taskloop

Semantics

The **grainzise** clause specifies that the number of logical iterations assigned to each generated task is greater than or equal to the minimum of the value of the *grain-size* expression and the number of logical iterations, but less than two times the value of the *grain-size* expression. If *prescriptiveness* is specified as **strict**, the number of logical iterations assigned to each generated task is equal to the value of the *grain-size* expression, except for the generated task that contains the sequentially last iteration, which may have fewer iterations.

Restrictions

Restrictions to the **grainsize** clause are as follows:

- None of the associated loops may be non-rectangular loops.

Cross References

- **taskloop** construct, see Section 12.6.

12.6.2 num_tasks Clause

Name:	Properties:
num_tasks	unique

Arguments:

Name	Type	Properties
num-tasks	Expression of type integer	default

Modifiers:

Name	Modifies	Type	Properties
prescriptiveness	*num-tasks*	**Keyword: strict**	unique

Directives:

taskloop

Semantics

The **num_tasks** clause specifies that the **taskloop** construct create as many tasks as the minimum of the *num-tasks* expression and the number of logical iterations. Each task must have at least one logical iteration. If *prescriptiveness* is specified as **strict** for a task loop with N logical iterations, the logical iterations are partitioned in a balanced manner and each partition is assigned, in order, to a generated task. The partition size is $\lceil N/\textit{num-tasks} \rceil$ until the number of remaining iterations divides the number of remaining tasks evenly, at which point the partition size becomes $\lfloor N/\textit{num-tasks} \rfloor$.

Restrictions

Restrictions to the **num_tasks** clause are as follows:

- None of the associated loops may be non-rectangular loops.

Cross References

- **taskloop** construct, see Section 12.6.

12.7 taskyield Construct

Name: **taskyield**	Association: none
Category: executable	**Properties:** default

Binding

A **taskyield** region binds to the current task region. The binding thread set of the **taskyield** region is the current team.

Semantics

The **taskyield** region includes an explicit task scheduling point in the current task region.

Cross References

- Task scheduling, see Section 12.9.

12.8 Initial Task

Execution Model Events

No events are associated with the implicit parallel region in each initial thread.

The *initial-thread-begin* event occurs in an initial thread after the OpenMP runtime invokes the tool initializer but before the initial thread begins to execute the first OpenMP region in the initial task.

The *initial-task-begin* event occurs after an *initial-thread-begin* event but before the first OpenMP region in the initial task begins to execute.

The *initial-task-end* event occurs before an *initial-thread-end* event but after the last OpenMP region in the initial task finishes execution.

The *initial-thread-end* event occurs as the final event in an initial thread at the end of an initial task immediately prior to invocation of the tool finalizer.

Tool Callbacks

A thread dispatches a registered **ompt_callback_thread_begin** callback for the *initial-thread-begin* event in an initial thread. The callback occurs in the context of the initial thread. The callback has type signature **ompt_callback_thread_begin_t**. The callback receives **ompt_thread_initial** as its *thread_type* argument.

A thread dispatches a registered **ompt_callback_implicit_task** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *initial-task-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_implicit_task** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of an *initial-task-end* event in that thread. The callbacks occur in the context of the initial task and have type signature **ompt_callback_implicit_task_t**. In the dispatched callback, **(***flag* **& ompt_task_initial)** always evaluates to *true*.

A thread dispatches a registered **ompt_callback_thread_end** callback for the *initial-thread-end* event in that thread. The callback occurs in the context of the thread. The callback has type signature **ompt_callback_thread_end_t**. The implicit parallel region does not dispatch a **ompt_callback_parallel_end** callback; however, the implicit parallel region can be finalized within this **ompt_callback_thread_end** callback.

Cross References

- **ompt_callback_implicit_task_t**, see Section 19.5.2.11.
- **ompt_callback_parallel_begin_t**, see Section 19.5.2.3.
- **ompt_callback_parallel_end_t**, see Section 19.5.2.4.
- **ompt_callback_thread_begin_t**, see Section 19.5.2.1.
- **ompt_callback_thread_end_t**, see Section 19.5.2.2.
- **ompt_task_initial**, see Section 19.4.4.19.
- **ompt_thread_initial**, see Section 19.4.4.10.

12.9 Task Scheduling

Whenever a thread reaches a task scheduling point, the implementation may cause it to perform a task switch, beginning or resuming execution of a different task bound to the current team. Task scheduling points are implied at the following locations:

- during the generation of an explicit task;
- the point immediately following the generation of an explicit task;
- after the point of completion of the structured block associated with a task;
- in a **taskyield** region;

- in a **taskwait** region;
- at the end of a **taskgroup** region;
- in an implicit barrier region;
- in an explicit **barrier** region;
- during the generation of a **target** region;
- the point immediately following the generation of a **target** region;
- at the beginning and end of a **target data** region;
- in a **target update** region;
- in a **target enter data** region;
- in a **target exit data** region;
- in the **omp_target_memcpy** routine;
- in the **omp_target_memcpy_async** routine;
- in the **omp_target_memcpy_rect** routine; and
- in the **omp_target_memcpy_rect_async** routine.

When a thread encounters a task scheduling point it may do one of the following, subject to the *Task Scheduling Constraints* (below):

- begin execution of a tied task bound to the current team;
- resume any suspended task region, bound to the current team, to which it is tied;
- begin execution of an untied task bound to the current team; or
- resume any suspended untied task region bound to the current team.

If more than one of the above choices is available, which one is chosen is unspecified.

Task Scheduling Constraints are as follows:

1. Scheduling of new tied tasks is constrained by the set of task regions that are currently tied to the thread and that are not suspended in a barrier region. If this set is empty, any new tied task may be scheduled. Otherwise, a new tied task may be scheduled only if it is a descendant task of every task in the set.
2. A dependent task shall not start its execution until its task dependences are fulfilled.
3. A task shall not be scheduled while any task with which it is mutually exclusive has been scheduled but has not yet completed.

4. When an explicit task is generated by a construct that contains an **if** clause for which the expression evaluated to *false*, and the previous constraints are already met, the task is executed immediately after generation of the task.

A program that relies on any other assumption about task scheduling is non-conforming.

Note – Task scheduling points dynamically divide task regions into parts. Each part is executed uninterrupted from start to end. Different parts of the same task region are executed in the order in which they are encountered. In the absence of task synchronization constructs, the order in which a thread executes parts of different schedulable tasks is unspecified.

A program must behave correctly and consistently with all conceivable scheduling sequences that are compatible with the rules above.

For example, if **threadprivate** storage is accessed (explicitly in the source code or implicitly in calls to library routines) in one part of a task region, its value cannot be assumed to be preserved into the next part of the same task region if another schedulable task exists that modifies it.

As another example, if a lock acquire and release happen in different parts of a task region, no attempt should be made to acquire the same lock in any part of another task that the executing thread may schedule. Otherwise, a deadlock is possible. A similar situation can occur when a **critical** region spans multiple parts of a task and another schedulable task contains a **critical** region with the same name.

The use of threadprivate variables and the use of locks or critical sections in an explicit task with an **if** clause must take into account that when the **if** clause evaluates to *false*, the task is executed immediately, without regard to *Task Scheduling Constraint* 2.

Execution Model Events

The *task-schedule* event occurs in a thread when the thread switches tasks at a task scheduling point; no event occurs when switching to or from a merged task.

Tool Callbacks

A thread dispatches a registered **ompt_callback_task_schedule** callback for each occurrence of a *task-schedule* event in the context of the task that begins or resumes. This callback has the type signature **ompt_callback_task_schedule_t**. The argument *prior_task_status* is used to indicate the cause for suspending the prior task. This cause may be the completion of the prior task region, the encountering of a **taskyield** construct, or the encountering of an active cancellation point.

Cross References

- **ompt_callback_task_schedule_t**, see Section 19.5.2.10.

13 Device Directives and Clauses

This chapter defines constructs and concepts related to device execution.

13.1 `device_type` Clause

Name:	**Properties:**
`device_type`	unique

Arguments:

Name	Type	Properties
device-type-description	**Keyword: `any`, `host`, `nohost`**	default

Directives:

`begin declare target`, **`declare target`**

Semantics

The **`device_type`** clause specifies if a version of the procedure or variable should be made available on the host device, non-host devices or both the host device and non-host devices. If **`host`** is specified then only a host device version of the procedure or variable is made available. If **`any`** is specified then both host device and non-host device versions of the procedure or variable are made available. If **`nohost`** is specified for a procedure then only non-host device versions of the procedure are made available. If **`nohost`** is specified for a variable then that variable is not available on the host device. If the **`device_type`** clause is not specified, the behavior is as if the **`device_type`** clause appears with **`any`** specified.

Cross References

- **`begin declare target`** directive, see Section 7.8.2.
- **`declare target`** directive, see Section 7.8.1.

13.2 `device` Clause

Name:	**Properties:**
`device`	unique

Arguments:

Name	Type	Properties
device-description	Expression of type integer	default

Modifiers:

Name	Modifies	Type	Properties
device-modifier	*device-description*	**Keyword:** **ancestor**, **device_num**	default

Directives:

target, **target data**, **target enter data**, **target exit data**, **target update**

Semantics

The **device** clause identifies the target device that is associated with a device construct.

If **device_num** is specified as the *device-modifier*, the *device-description* specifies the device number of the target device. If *device-modifier* does not appear in the clause, the behavior of the clause is as if *device-modifier* is **device_num**. If the *device-description* evaluates to **omp_invalid_device**, runtime error termination is performed.

If **ancestor** is specified as the *device-modifier*, the *device-description* specifies the number of target nesting level of the target device. Specifically, if the *device-description* evaluates to 1, the target device is the parent device of the enclosing **target** region. If the construct on which the **device** clause appears is not encountered in a **target** region, the current device is treated as the parent device.

Unless otherwise specified, for directives that accept the **device** clause, if no **device** clause is present, the behavior is as if the **device** clause appears without a *device-modifier* and with a *device-description* that evaluates to the value of the *default-device-var* ICV.

Restrictions

- The **ancestor** *device-modifier* must not appear on the **device** clause on any directive other than the **target** construct.
- If the **ancestor** *device-modifier* is specified, the **device-description** must evaluate to 1 and a **requires** directive with the **reverse_offload** clause must be specified;
- If the **device_num** *device-modifier* is specified and *target-offload-var* is not **mandatory**, *device-description* must evaluate to a conforming device number.

Cross References

- *default-device-var*, see Section 2.1.
- **omp_get_num_devices** routine, see Section 18.7.4.
- **target** construct, see Section 13.8.
- **target data** construct, see Section 13.5.
- **target enter data** construct, see Section 13.6.
- **target exit data** construct, see Section 13.7.
- **target update** construct, see Section 13.9.

13.3 `thread_limit` Clause

Name: `thread_limit` | **Properties:** unique

Arguments:

Name	Type	Properties
threadlim	Expression of type integer	positive

Directives:
`parallel`

Semantics

As described in Section 2.4.1, some constructs limit the number of threads that may participate in a contention group initiated by each team by setting the value of the *thread-limit-var* ICV for the initial task to an implementation defined value greater than zero. If the **`thread_limit`** clause is specified, the number of threads will be less than or equal to *threadlim*. Otherwise, if the *teams-thread-limit-var* ICV is greater than zero, the effect is as if the *thread_limit* clause was specified with a *threadlim* that evaluates to an implementation defined value less than or equal to the *teams-thread-limit-var* ICV.

Cross References

- **`target`** construct, see Section 13.8.
- **`team`** construct, see Section 10.2.

13.4 Device Initialization

Execution Model Events

The *device-initialize* event occurs in a thread that encounters the first **`target`**, **`target data`**, or **`target enter data`** construct or a device memory routine that is associated with a particular target device after the thread initiates initialization of OpenMP on the device and the device's OpenMP initialization, which may include device-side tool initialization, completes.

The *device-load* event for a code block for a target device occurs in some thread before any thread executes code from that code block on that target device.

The *device-unload* event for a target device occurs in some thread whenever a code block is unloaded from the device.

The *device-finalize* event for a target device that has been initialized occurs in some thread before an OpenMP implementation shuts down.

Tool Callbacks

A thread dispatches a registered **ompt_callback_device_initialize** callback for each occurrence of a *device-initialize* event in that thread. This callback has type signature **ompt_callback_device_initialize_t**.

A thread dispatches a registered **ompt_callback_device_load** callback for each occurrence of a *device-load* event in that thread. This callback has type signature **ompt_callback_device_load_t**.

A thread dispatches a registered **ompt_callback_device_unload** callback for each occurrence of a *device-unload* event in that thread. This callback has type signature **ompt_callback_device_unload_t**.

A thread dispatches a registered **ompt_callback_device_finalize** callback for each occurrence of a *device-finalize* event in that thread. This callback has type signature **ompt_callback_device_finalize_t**.

Restrictions

Restrictions to OpenMP device initialization are as follows:

- No thread may offload execution of an OpenMP construct to a device until a dispatched **ompt_callback_device_initialize** callback completes.
- No thread may offload execution of an OpenMP construct to a device after a dispatched **ompt_callback_device_finalize** callback occurs.

Cross References

- **ompt_callback_device_finalize_t**, see Section 19.5.2.20.
- **ompt_callback_device_initialize_t**, see Section 19.5.2.19.
- **ompt_callback_device_load_t**, see Section 19.5.2.21.
- **ompt_callback_device_unload_t**, see Section 19.5.2.22.

13.5 target data Construct

Name: target data **Category:** executable	**Association:** block **Properties:** device, device-affecting, data-mapping, map-entering, map-exiting, mapping-only

Clauses:
device, **if**, **map**, **use_device_addr**, **use_device_ptr**

Binding

The binding task set for a **target data** region is the generating task. The **target data** region binds to the region of the generating task.

Semantics

The **target data** construct maps variables to a device data environment. When a **target data** construct is encountered, the encountering task executes the region. When an **if** clause is present and the **if** clause expression evaluates to *false*, the target device is the host. Variables are mapped for the extent of the region, according to any data-mapping attribute clauses, from the data environment of the encountering task to the device data environment.

If one or more **map** clauses are present, the list item conversions that are performed for any **use_device_ptr** or **use_device_addr** clause occur after all variables are mapped on entry to the region according to those **map** clauses.

Execution Model Events

The events associated with entering a **target data** region are the same events as associated with a **target enter data** construct, as described in Section 13.6.

The events associated with exiting a **target data** region are the same events as associated with a **target exit data** construct, as described in Section 13.7.

Tool Callbacks

The tool callbacks dispatched when entering a **target data** region are the same as the tool callbacks dispatched when encountering a **target enter data** construct, as described in Section 13.6.

The tool callbacks dispatched when exiting a **target data** region are the same as the tool callbacks dispatched when encountering a **target exit data** construct, as described in Section 13.7.

Restrictions

Restrictions to the **target data** construct are as follows:

- A *map-type* in a **map** clause must be **to**, **from**, **tofrom** or **alloc**.
- At least one **map**, **use_device_addr** or **use_device_ptr** clause must appear on the directive.

Cross References

- **device** clause, see Section 13.2.
- **if** clause, see Section 3.4.
- **map** clause, see Section 5.8.2.
- **use_device_addr** clause, see Section 5.8.6.
- **use_device_ptr** clause, see Section 5.8.4.

13.6 `target enter data` Construct

Name: `target enter data` **Category:** executable	**Association:** none **Properties:** parallelism-generating, task-generating, device, device-affecting, data-mapping, map-entering, mapping-only

Clauses:
`depend`, **`device`**, **`if`**, **`map`**, **`nowait`**

Binding

The binding task set for a **`target enter data`** region is the generating task, which is the *target task* generated by the **`target enter data`** construct. The **`target enter data`** region binds to the corresponding *target task* region.

Semantics

When a **`target enter data`** construct is encountered, the list items are mapped to the device data environment according to the **`map`** clause semantics. The **`target enter data`** construct generates a *target task*. The generated task region encloses the **`target enter data`** region. If a **`depend`** clause is present, it is associated with the *target task*. If the **`nowait`** clause is present, execution of the *target task* may be deferred. If the **`nowait`** clause is not present, the *target task* is an included task.

All clauses are evaluated when the **`target enter data`** construct is encountered. The data environment of the *target task* is created according to the data-mapping attribute clauses on the **`target enter data`** construct, per-data environment ICVs, and any default data-sharing attribute rules that apply to the **`target enter data`** construct. If a variable or part of a variable is mapped by the **`target enter data`** construct, the variable has a default data-sharing attribute of shared in the data environment of the *target task*.

Assignment operations associated with mapping a variable (see Section 5.8.2) occur when the *target task* executes.

When an **`if`** clause is present and the **`if`** clause expression evaluates to *false*, the target device is the host.

Execution Model Events

Events associated with a *target task* are the same as for the **`task`** construct defined in Section 12.5.

The *target-enter-data-begin* event occurs after creation of the target task and completion of all predecessor tasks that are not target tasks for the same device. The *target-enter-data-begin* event is a *target-task-begin* event.

The *target-enter-data-end* event occurs after all other events associated with the **`target enter data`** construct.

Tool Callbacks

Callbacks associated with events for *target tasks* are the same as for the **task** construct defined in Section 12.5; **(***flags* **& ompt_task_target)** always evaluates to *true* in the dispatched callback.

A thread dispatches a registered **ompt_callback_target** or **ompt_callback_target_emi** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_target_enter_data** or **ompt_target_enter_data_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-enter-data-begin* event in that thread in the context of the target task on the host. Similarly, a thread dispatches a registered **ompt_callback_target** or **ompt_callback_target_emi** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_target_enter_data** or **ompt_target_enter_data_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-enter-data-end* event in that thread in the context of the target task on the host. These callbacks have type signature **ompt_callback_target_t** or **ompt_callback_target_emi_t**, respectively.

Restrictions

Restrictions to the **target enter data** construct are as follows:

- At least one **map** clause must appear on the directive.
- All **map** clauses must be *map-entering*.

Cross References

- **device** clause, see Section 13.2.
- **if** clause, see Section 3.4.
- **ompt_callback_target_t** and **ompt_callback_target_emi_t** callback type, see Section 19.5.2.26.
- Task scheduling constraints, see Section 12.9.
- **map** clause, see Section 5.8.2.
- **nowait** clause, see Section 15.6.
- **target data** construct, see Section 13.5.
- **target exit data** construct, see Section 13.7.
- **task** construct, see Section 12.5.

13.7 `target exit data` Construct

Name: `target exit data` **Category:** executable	**Association:** none **Properties:** parallelism-generating, task-generating, device, device-affecting, data-mapping, map-exiting, mapping-only

Clauses:
depend, **device**, **if**, **map**, **nowait**

Binding

The binding task set for a **target exit data** region is the generating task, which is the *target task* generated by the **target exit data** construct. The **target exit data** region binds to the corresponding *target task* region.

Semantics

When a **target exit data** construct is encountered, the list items in the **map** clauses are unmapped from the device data environment according to the **map** clause semantics. The **target exit data** construct generates a *target task*. The generated taskregion encloses the **target exit data** region. If a **depend** clause is present, it is associated with the *target task*. If the **nowait** clause is present, execution of the *target task* may be deferred. If the **nowait** clause is not present, the *target task* is an included task.

All clauses are evaluated when the **target exit data** construct is encountered. The data environment of the *target task* is created according to the data-mapping attribute clauses on the **target exit data** construct, per-data environment ICVs, and any default data-sharing attribute rules that apply to the **target exit data** construct. If a variable or part of a variable is mapped by the **target exit data** construct, the variable has a default data-sharing attribute of shared in the data environment of the *target task*.

Assignment operations associated with mapping a variable (see Section 5.8.2) occur when the *target task* executes.

When an **if** clause is present and the **if** clause expression evaluates to *false*, the target device is the host.

Execution Model Events

Events associated with a *target task* are the same as for the **task** construct defined in Section 12.5.

The *target-exit-data-begin* event occurs after creation of the target task and completion of all predecessor tasks that are not target tasks for the same device. The *target-exit-data-begin* event is a *target-task-begin* event.

The *target-exit-data-end* event occurs after all other events associated with the **target exit data** construct.

Tool Callbacks

Callbacks associated with events for *target tasks* are the same as for the **task** construct defined in Section 12.5; **(***flags* **& ompt_task_target)** always evaluates to *true* in the dispatched callback.

A thread dispatches a registered **ompt_callback_target** or **ompt_callback_target_emi** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_target_exit_data** or **ompt_target_exit_data_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-exit-data-begin* event in that thread in the context of the target task on the host. Similarly, a thread dispatches a registered **ompt_callback_target** or **ompt_callback_target_emi** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_target_exit_data** or **ompt_target_exit_data_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-exit-data-end* event in that thread in the context of the target task on the host. These callbacks have type signature **ompt_callback_target_t** or **ompt_callback_target_emi_t**, respectively.

Restrictions

Restrictions to the **target exit data** construct are as follows:

- At least one **map** clause must appear on the directive.
- All **map** clauses must be a *map-exiting*.

Cross References

- **device** clause, see Section 13.2.
- **if** clause, see Section 3.4.
- **ompt_callback_target_t** and **ompt_callback_target_emi_t** callback type, see Section 19.5.2.26.
- **task scheduling constraints**, see Section 12.9.
- **map** clause, see Section 5.8.2.
- **nowait** clause, see Section 15.6.
- **target data**, see Section 13.5.
- **target enter data**, see Section 13.6.
- **task**, see Section 12.5.

13.8 `target` Construct

Name: `target` **Category:** executable	**Association:** block **Properties:** parallelism-generating, thread-limiting, exception-aborting, task-generating, device, device-affecting, data-mapping, map-entering, map-exiting, context-matching

Clauses:
`allocate`, `defaultmap`, `depend`, `device`, `firstprivate`, `has_device_addr`, `if`, `in_reduction`, `is_device_ptr`, `map`, `nowait`, `private`, `thread_limit`, `uses_allocators`

Binding

The binding task set for a **`target`** region is the generating task, which is the *target task* generated by the **`target`** construct. The **`target`** region binds to the corresponding *target task* region.

Semantics

The **`target`** construct provides a superset of the functionality provided by the **`target data`** directive, except for the **`use_device_ptr`** and **`use_device_addr`** clauses. The functionality added to the **`target`** directive is the inclusion of an executable region to be executed on a device. The **`target`** construct generates a *target task*. The generated task region encloses the **`target`** region. If a **`depend`** clause is present, it is associated with the *target task*. The **`device`** clause determines the device on which the **`target`** region executes. If the **`nowait`** clause is present, execution of the *target task* may be deferred. If the **`nowait`** clause is not present, the *target task* is an included task.

All clauses are evaluated when the **`target`** construct is encountered. The data environment of the *target task* is created according to the data-sharing and data-mapping attribute clauses on the **`target`** construct, per-data environment ICVs, and any default data-sharing attribute rules that apply to the **`target`** construct. If a variable or part of a variable is mapped by the **`target`** construct and does not appear as a list item in an **`in_reduction`** clause on the construct, the variable has a default data-sharing attribute of shared in the data environment of the *target task*. Assignment operations associated with mapping a variable (see Section 5.8.2) occur when the *target task* executes.

If the **`device`** clause is specified with the **`ancestor`** *device-modifier*, the encountering thread waits for completion of the **`target`** region on the parent device before resuming. For any list item that appears in a **`map`** clause on the same construct, if the corresponding list item exists in the device data environment of the parent device, it is treated as if it has a reference count of positive infinity.

When an **`if`** clause is present and the **`if`** clause expression evaluates to *false*, the effect is as if a **`device`** clause that specifies **`omp_initial_device`** as the device number is present, regardless of any other **`device`** clause on the directive.

If a procedure is explicitly or implicitly referenced in a **target** construct that does not specify a **device** clause in which the **ancestor** *device-modifier* appears then that procedure is treated as if its name had appeared in an **enter** clause on a declare target directive.

If a variable with static storage duration is declared in a **target** construct that does not specify a **device** clause in which the **ancestor** *device-modifier* appears then the named variable is treated as if it had appeared in a **enter** clause on a declare target directive.

C / C++

If a list item in a **map** clause has a base pointer and it is a scalar variable with a predetermined data-sharing attribute of firstprivate (see Section 5.1.1), then on entry to the **target** region:

- If the list item is not a zero-length array section, the corresponding private variable is initialized such that the corresponding list item in the device data environment can be accessed through the pointer in the **target** region.
- If the list item is a zero-length array section , the corresponding private variable is initialized according to Section 5.8.8.

C / C++

Fortran

When an internal procedure is called in a **target** region, any references to variables that are host associated in the procedure have unspecified behavior.

Fortran

Execution Model Events

Events associated with a *target task* are the same as for the **task** construct defined in Section 12.5.

Events associated with the *initial task* that executes the **target** region are defined in Section 12.8.

The *target-submit-begin* event occurs prior to initiating creation of an initial task on a target device for a **target** region.

The *target-submit-end* event occurs after initiating creation of an initial task on a target device for a **target** region.

The *target-begin* event occurs after creation of the target task and completion of all predecessor tasks that are not target tasks for the same device. The *target-begin* event is a *target-task-begin* event.

The *target-end* event occurs after all other events associated with the **target** construct.

Tool Callbacks

Callbacks associated with events for *target tasks* are the same as for the **task** construct defined in Section 12.5; **(***flags* **& ompt_task_target)** always evaluates to *true* in the dispatched callback.

A thread dispatches a registered **ompt_callback_target** or **ompt_callback_target_emi** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_target** or **ompt_target_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-begin* event in that thread in the context of the target task on the host. Similarly, a thread dispatches a registered **ompt_callback_target** or **ompt_callback_target_emi** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_target** or **ompt_target_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-end* event in that thread in the context of the target task on the host. These callbacks have type signature **ompt_callback_target_t** or **ompt_callback_target_emi_t**, respectively.

A thread dispatches a registered **ompt_callback_target_submit_emi** callback with **ompt_scope_begin** as its endpoint argument for each occurrence of a *target-submit-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_target_submit_emi** callback with **ompt_scope_end** as its endpoint argument for each occurrence of a *target-submit-end* event in that thread. These callbacks have type signature **ompt_callback_target_submit_emi_t**.

A thread dispatches a registered **ompt_callback_target_submit** callback for each occurrence of a *target-submit-begin* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_submit_t**.

Restrictions

Restrictions to the **target** construct are as follows:

- Device-affecting constructs, other than **target** constructs for which the **ancestor** *device-modifier* is specified, must not be encountered during execution of a **target** region.
- The result of an **omp_set_default_device**, **omp_get_default_device**, or **omp_get_num_devices** routine called within a **target** region is unspecified.
- The effect of an access to a threadprivate variable in a **target** region is unspecified.
- If a list item in a **map** clause is a structure element, any other element of that structure that is referenced in the **target** construct must also appear as a list item in a **map** clause.
- A list item in a data-sharing attribute clause that is specified on a **target** construct must not have the same base variable as a list item in a **map** clause on the construct.
- A variable referenced in a **target** region but not the **target** construct that is not declared in the **target** region must appear in a declare target directive.
- A *map-type* in a **map** clause must be **to**, **from**, **tofrom** or **alloc**.

- If a **device** clause is specified with the **ancestor** *device-modifier*, only the **device**, **firstprivate**, **private**, **defaultmap**, and **map** clauses may appear on the construct and no OpenMP constructs or calls to OpenMP API runtime routines are allowed inside the corresponding **target** region.
- Memory allocators that do not appear in a **uses_allocators** clause cannot appear as an allocator in an **allocate** clause or be used in the **target** region unless a **requires** directive with the **dynamic_allocators** clause is present in the same compilation unit.
- Any IEEE floating-point exception status flag, halting mode, or rounding mode set prior to a **target** region is unspecified in the region.
- Any IEEE floating-point exception status flag, halting mode, or rounding mode set in a **target** region is unspecified upon exiting the region.
- A program must not rely on the value of a function address in a **target** region except for assignments, comparisons to zero and indirect calls.

C / C++

- An attached pointer must not be modified in a **target** region.

C / C++

C++

- The run-time type information (RTTI) of an object can only be accessed from the device on which it was constructed.
- Invoking a virtual member function of an object on a device other than the device on which the object was constructed results in unspecified behavior, unless the object is accessible and was constructed on the host device.
- If an object of polymorphic class type is destructed, virtual member functions of any previously existing corresponding objects in other device data environments must not be invoked.

C++

Fortran

- An attached pointer that is associated with a given pointer target must not become associated with a different pointer target in a **target** region.
- If a list item in a **map** clause is an array section, and the array section is derived from a variable with a **POINTER** or **ALLOCATABLE** attribute then the behavior is unspecified if the corresponding list item's variable is modified in the region.
- A reference to a coarray that is encountered on a non-host device must not be coindexed or appear as an actual argument to a procedure where the corresponding dummy argument is a coarray.
- If the allocation status of a mapped variable that has the **ALLOCATABLE** attribute is unallocated on entry to a **target** region, the allocation status of the corresponding variable in the device data environment must be unallocated upon exiting the region.

- If the allocation status of a mapped variable that has the **ALLOCATABLE** attribute is allocated on entry to a **target** region, the allocation status and shape of the corresponding variable in the device data environment may not be changed, either explicitly or implicitly, in the region after entry to it.
- If the association status of a list item with the **POINTER** attribute that appears in a **map** clause on the construct is associated upon entry to the **target** region, the list item must be associated with the same pointer target upon exit from the region.
- If the association status of a list item with the **POINTER** attribute that appears in a **map** clause on the construct is disassociated upon entry to the **target** region, the list item must be disassociated upon exit from the region.
- If the association status of a list item with the **POINTER** attribute that appears in a **map** clause on the construct is disassociated or undefined on entry to the **target** region and if the list item is associated with a pointer target inside the **target** region, the pointer association status must become disassociated before the end the region.
- A program must not rely on the association status of a procedure pointer in a **target** region except for calls to the **ASSOCIATED** inquiry function without the optional *proc-target* argument, pointer assignments and indirect calls.

Fortran

Cross References

- **device** clause, see Section 13.2.
- **if** clause, see Section 3.4.
- **ompt_callback_target_t** or **ompt_callback_target_emi_t** callback type, see Section 19.5.2.26.
- **ompt_callback_target_submit_t** or **ompt_callback_target_submit_emi_t** callback type, Section 19.5.2.28.
- **uses_allocators** clause, see Section 6.9.
- Data-Mapping Attribute Rules and Clauses, see Section 5.8.
- **private** and **firstprivate** clauses, see Section 5.4.
- **omp_alloctrait_t** and **omp_alloctrait** types, see Section 18.13.1.
- **task** scheduling constraints, see Section 12.9
- **has_device_addr** clause, see Section 5.8.5.
- **is_device_ptr** clause, see Section 5.8.3.
- **nowait** clause, see Section 15.6.
- **omp_get_default_allocator** routine, see Section 18.13.5.

- **omp_set_default_allocator** routine, see Section 18.13.4.
- **target data** construct, see Section 13.5.
- **task** construct, see Section 12.5.

13.9 target update Construct

Name: target update **Category:** executable	**Association:** none **Properties:** parallelism-generating, task-generating, device, device-affecting

Clauses:
depend, **device**, **from**, **if**, **nowait**, **to**

Clause set:

Properties: required	**Members:** to, from

Binding

The binding task set for a **target update** region is the generating task, which is the *target task* generated by the **target update** construct. The **target update** region binds to the corresponding *target task* region.

Semantics

The **target update** directive makes the corresponding list items in the device data environment consistent with their original list items, according to the specified **to** and **from** clauses. The **target update** construct generates a *target task*. The generated task region encloses the **target update** region. If a **depend** clause is present, it is associated with the *target task*. If the **nowait** clause is present, execution of the *target task* may be deferred. If the **nowait** clause is not present, the *target task* is an included task.

All clauses are evaluated when the **target update** construct is encountered. The data environment of the *target task* is created according to **to** and **from** clauses on the **target update** construct, per-data environment ICVs, and any default data-sharing attribute rules that apply to the **target update** construct. If a variable or part of a variable is a list item in a **to** or **from** clause on the **target update** construct, the variable has a default data-sharing attribute of **shared** in the data environment of the *target task*.

Assignment operations associated with any motion clauses occur when the *target task* executes. When an **if** clause is present and the **if** clause expression evaluates to *false*, no assignments occur.

Execution Model Events

Events associated with a *target task* are the same as for the **task** construct defined in Section 12.5.

The *target-update-begin* event occurs after creation of the target task and completion of all predecessor tasks that are not target tasks for the same device.

The *target-update-end* event occurs after all other events associated with the **target update** construct.

The *target-data-op-begin* event occurs in the **target update** region before a thread initiates a data operation on the target device.

The *target-data-op-end* event occurs in the **target update** region after a thread initiates a data operation on the target device.

Tool Callbacks

Callbacks associated with events for *target tasks* are the same as for the **task** construct defined in Section 12.5; **(***flags* **& ompt_task_target)** always evaluates to *true* in the dispatched callback.

A thread dispatches a registered **ompt_callback_target** or **ompt_callback_target_emi** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_target_update** or **ompt_target_update_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-update-begin* event in that thread in the context of the target task on the host. Similarly, a thread dispatches a registered **ompt_callback_target** or **ompt_callback_target_emi** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_target_update** or **ompt_target_update_nowait** if the **nowait** clause is present as its *kind* argument for each occurrence of a *target-update-end* event in that thread in the context of the target task on the host. These callbacks have type signature **ompt_callback_target_t** or **ompt_callback_target_emi_t**, respectively.

A thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_begin** as its endpoint argument for each occurrence of a *target-data-op-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_end** as its endpoint argument for each occurrence of a *target-data-op-end* event in that thread. These callbacks have type signature **ompt_callback_target_data_op_emi_t**.

A thread dispatches a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-op-end* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**.

Cross References

- **device** clause, see Section 13.2.
- **if** clause, see Section 3.4.

- **ompt_callback_target_t** or **ompt_callback_target_emi_t** callback type, see Section 19.5.2.26.
- **ompt_callback_task_create_t**, see Section 19.5.2.7.
- Task scheduling constraints, see Section 12.9.
- **from** clause, see Section 5.9.2.
- **nowait** clause, see Section 15.6.
- **target data** construct, see Section 13.5.
- **task** construct, see Section 12.5.
- **to** clause, see Section 5.9.1.

14 Interoperability

An OpenMP implementation may interoperate with one or more foreign runtime environments through the use of the **interop** construct that is described in this chapter, the **interop** operation for a declared variant function and the interoperability routines that are available through the OpenMP Runtime API.

C / C++

The implementation must provide *foreign-runtime-id* values that are enumerators of type **omp_interop_fr_t** and that correspond to the supported foreign runtime environments.

C / C++

Fortran

The implementation must provide *foreign-runtime-id* values that are named integer constants with kind **omp_interop_fr_kind** and that correspond to the supported foreign runtime environments.

Fortran

Each *foreign-runtime-id* value provided by an implementation will be available as **omp_ifr_*name***, where *name* is the name of the foreign runtime environment. Available names include those that are listed in the *OpenMP Additional Definitions* document; implementation-defined names may also be supported. The value of **omp_ifr_last** is defined as one greater than the value of the highest supported *foreign-runtime-id* value that is listed in the aforementioned document.

Cross References

- Interoperability routines, see Section 18.12.
- **declare variant** directive, see Section 7.5.

14.1 interop Construct

Name: **interop** **Category:** executable	**Association:** none **Properties:** device

Clauses:
depend, **destroy**, **device**, **init**, **nowait**, **use**

Clause set: action-clause

Properties: required	**Members:** init, destroy, use

Binding

The binding task set for an **interop** region is the generating task. The **interop** region binds to the region of the generating task.

Semantics

The **interop** construct retrieves interoperability properties from the OpenMP implementation to enable interoperability with foreign execution contexts. When an **interop** construct is encountered, the encountering task executes the region.

For each *action-clause*, the *interop-type* set is the set of *interop-type* modifiers specified for the clause if the clause is **init** or for the *init* clause that initialized the *interop-var* that is specified for the clause if the clause is not **init**.

If the *interop-type* set includes **targetsync**, an empty *mergeable task* is generated. If the **nowait** clause is not present on the construct then the task is also an *included task*. Any **depend** clauses that are present on the construct apply to the generated task.

The **interop** construct ensures an ordered execution of the generated task relative to foreign tasks executed in the foreign execution context through the foreign synchronization object that is accessible through the **targetsync** property. When the creation of the foreign task precedes the encountering of an **interop** construct in happens before order (see Section 1.4.5), the foreign task must complete execution before the generated task begins execution. Similarly, when the creation of a foreign task follows the encountering of an **interop** construct in happens before order, the foreign task must not begin execution until the generated task completes execution. No ordering is imposed between the encountering thread and either foreign tasks or OpenMP tasks by the **interop** construct.

If the *interop-type* set does not include **targetsync**, the **nowait** clause has no effect.

Restrictions

Restrictions to the **interop** construct are as follows:

- A **depend** clause can only appear on the directive if the *interop-type* includes **targetsync**.
- Each *interop-var* may be specified for at most one *action-clause* of each **interop** construct.

Cross References

- Interoperability routines, see Section 18.12.
- **destroy** clause, see Section 3.5.
- **depend** clause, see Section 15.9.5.
- **init** clause, see Section 14.1.2.
- **use** clause, see Section 14.1.3.

14.1.1 OpenMP Foreign Runtime Identifiers

An OpenMP foreign runtime identifier, *foreign-runtime-id*, is a base language string literal or a compile-time constant OpenMP integer expression. Allowed values for *foreign-runtime-id* include the names (as string literals) and integer values that the *OpenMP Additional Definitions* document specifies and the corresponding **omp_ifr_*name*** constants of OpenMP **interop_fr** type. Implementation-defined values for *foreign-runtime-id* may also be supported.

14.1.2 init Clause

Name:	**Properties:**
init	unique

Arguments:

Name	Type	Properties
interop-var	Variable of type omp_interop_t	default

Modifiers:

Name	Modifies	Type	Properties
interop-preference	Generic	Complex modifier: Keyword:**prefer_type** Arguments: Name:*preference_list* Type: OpenMP foreign runtime preference list Properties: default	unique, complex
interop-type	Generic	**Keyword: target, targetsync**	default

Directives:

interop

Semantics

The **init** clause specifies that *interop-var* is initialized to refer to the list of properties associated with any *interop-type*. For any *interop-type*, the properties **type**, **type_name**, **vendor**, **vendor_name** and **device_num** will be available. If the implementation cannot initialize *interop-var*, it is initialized to the value of **omp_interop_none**, which is defined to be zero.

The **targetsync** *interop-type* will additionally provide the **targetsync** property, which is the handle to a foreign synchronization object for enabling synchronization between OpenMP tasks and foreign tasks that execute in the foreign execution context.

The **target** *interop-type* will additionally provide the following properties:

- **device**, which will be a foreign device handle;

- **device_context**, which will be a foreign device context handle; and
- **platform**, which will be a handle to a foreign platform of the device.

If the **prefer_type** *interop-modifier* clause is specified, the first supported *foreign-runtime-id* in *preference-list* in left-to-right order is used. The *foreign-runtime-id* that is used if the implementation does not support any of the items in *preference-list* is implementation defined.

Restrictions

Restrictions to the **init** clause are as follows:

- Each *interop-type* may be specified at most once.
- *interop-var* must be non-const.

Cross References

- **interop** construct, see Section 14.1.
- OpenMP foreign runtime idenitifiers, see Section 14.1.1.

14.1.3 use Clause

Name:	**Properties:**
use	unique

Arguments:

Name	Type	Properties
interop-var	Variable of type omp_interop_t	default

Directives:

interop

Semantics

The **use** clause specifies the *interop-var* that is used for the effects of the directive on which the clause appears. However, *interop-var* is not initialized, destroyed or otherwise modified. The *interop-type* is inferred based on the *interop-type* used to initialize *interop-var*.

Cross References

- **interop** construct, see Section 14.1.

14.2 Interoperability Requirement Set

The *interoperability requirement set* of each task is a logical set of properties that can be added or removed by different directives. These properties can be queried by other constructs that have interoperability semantics.

A construct can add the following properties to the set:

- *depend*, which specifies that the construct requires enforcement of the synchronization relationship expressed by the *depend* clause;
- *nowait*, which specifies that the construct is asynchronous; and
- *is_device_ptr(list-item)*, which specifies that the *list-item* is a device pointer in the construct.

The following directives may add properties to the set:

- **`dispatch`**.

The following directives may remove properties from the set:

- **`declare variant`**.

Cross References

- **`declare variant`** directive, see Section 7.5.
- **`dispatch`** construct, see Section 7.6.

15 Synchronization Constructs and Clauses

A synchronization construct orders the completion of code executed by different threads. This ordering is imposed by synchronizing flush operations that are executed as part of the region that corresponds to the construct.

Synchronization through the use of synchronizing flush operations and atomic operations is described in Section 1.4.4 and Section 1.4.6. Section 15.8.6 defines the behavior of synchronizing flush operations that are implied at various other locations in an OpenMP program.

15.1 Synchronization Hints

Hints about the expected dynamic behavior or suggested implementation can be provided by the programmer to locks (by using the **omp_init_lock_with_hint** or **omp_init_nest_lock_with_hint** functions to initialize the lock), and to **atomic** and **critical** directives by using the **hint** clause. The effect of a hint does not change the semantics of the associated construct; if ignoring the hint changes the program semantics, the result is unspecified.

15.1.1 Synchronization Hint Type

Synchronization hints are specified with an OpenMP type that has the <*generic_name*> **sync_hint**. The C/C++ header file (**omp.h**) and the Fortran include file (**omp_lib.h**) and/or Fortran module file (**omp_lib**) define the valid hint constants. The valid constants must include the following, which can be extended with implementation-defined values:

C / C++

```
typedef enum omp_sync_hint_t {
  omp_sync_hint_none = 0x0,
  omp_lock_hint_none = omp_sync_hint_none,
  omp_sync_hint_uncontended = 0x1,
  omp_lock_hint_uncontended = omp_sync_hint_uncontended,
  omp_sync_hint_contended = 0x2,
  omp_lock_hint_contended = omp_sync_hint_contended,
  omp_sync_hint_nonspeculative = 0x4,
  omp_lock_hint_nonspeculative = omp_sync_hint_nonspeculative,
```

```c
  omp_sync_hint_speculative = 0x8,
  omp_lock_hint_speculative = omp_sync_hint_speculative
} omp_sync_hint_t;

typedef omp_sync_hint_t omp_lock_hint_t;
```

C / C++

Fortran

```fortran
integer, parameter :: omp_lock_hint_kind = omp_sync_hint_kind

integer (kind=omp_sync_hint_kind), &
  parameter :: omp_sync_hint_none = &
                   int(Z'0', kind=omp_sync_hint_kind)
integer (kind=omp_lock_hint_kind), &
  parameter :: omp_lock_hint_none = omp_sync_hint_none
integer (kind=omp_sync_hint_kind), &
  parameter :: omp_sync_hint_uncontended = &
                   int(Z'1', kind=omp_sync_hint_kind)
integer (kind=omp_lock_hint_kind), &
  parameter :: omp_lock_hint_uncontended = &
                   omp_sync_hint_uncontended
integer (kind=omp_sync_hint_kind), &
  parameter :: omp_sync_hint_contended = &
                   int(Z'2', kind=omp_sync_hint_kind)
integer (kind=omp_lock_hint_kind), &
  parameter :: omp_lock_hint_contended = &
                   omp_sync_hint_contended
integer (kind=omp_sync_hint_kind), &
  parameter :: omp_sync_hint_nonspeculative = &
                   int(Z'4', kind=omp_sync_hint_kind)
integer (kind=omp_lock_hint_kind), &
  parameter :: omp_lock_hint_nonspeculative = &
                   omp_sync_hint_nonspeculative
integer (kind=omp_sync_hint_kind), &
  parameter :: omp_sync_hint_speculative = &
                   int(Z'8', kind=omp_sync_hint_kind)
integer (kind=omp_lock_hint_kind), &
  parameter :: omp_lock_hint_speculative = &
                   omp_sync_hint_speculative
```

Fortran

The hints can be combined by using the **+** or **|** operators in C/C++ or the **+** operator in Fortran. Combining **omp_sync_hint_none** with any other hint is equivalent to specifying the other hint.

The intended meaning of each hint is:

- **omp_sync_hint_uncontended**: low contention is expected in this operation, that is, few threads are expected to perform the operation simultaneously in a manner that requires synchronization;
- **omp_sync_hint_contended**: high contention is expected in this operation, that is, many threads are expected to perform the operation simultaneously in a manner that requires synchronization;
- **omp_sync_hint_speculative**: the programmer suggests that the operation should be implemented using speculative techniques such as transactional memory; and
- **omp_sync_hint_nonspeculative**: the programmer suggests that the operation should not be implemented using speculative techniques such as transactional memory.

Note – Future OpenMP specifications may add additional hints to the **sync_hint** type. Implementers are advised to add implementation-defined hints starting from the most significant bit of the type and to include the name of the implementation in the name of the added hint to avoid name conflicts with other OpenMP implementations.

The OpenMP **sync_hint** and **lock_hint** types are synonyms for each other. The **lock_hint** type has been deprecated.

Restrictions

Restrictions to the synchronization hints are as follows:

- The hints **omp_sync_hint_uncontended** and **omp_sync_hint_contended** cannot be combined.
- The hints **omp_sync_hint_nonspeculative** and **omp_sync_hint_speculative** cannot be combined.

The restrictions for combining multiple values of **omp_sync_hint** apply equally to the corresponding values of **omp_lock_hint**, and expressions that mix the two types.

Cross References

- **hint** clause, see Section 15.1.2
- **omp_init_lock_with_hint** and **omp_init_nest_lock_with_hint**, see Section 18.9.2.

15.1.2 hint Clause

Name:	**Properties:**
hint	unique

Arguments:

Name	Type	Properties
hint-expr	Expression of type sync_hint	default

Directives:

atomic, **critical**

Semantics

The **hint** clause gives the implementation additional information about the expected runtime properties of the region that corresponds to the construct on which it appears and that can optionally be used to optimize the implementation. The presence of a **hint** clause does not affect the semantics of the construct. If no **hint** clause is specified for a construct that accepts it, the effect is as if **hint(omp_sync_hint_none)** had been specified.

Restrictions

- *hint-expr* must evaluate to a valid synchronization hint.

Cross References

- **atomic** construct, see Section 15.8.4
- **critical** construct, see Section 15.2.

15.2 critical Construct

Name: critical **Category:** executable	**Association:** block **Properties:** thread-limiting

Arguments: critical(*name*)

Name	Type	Properties
name	Identifier of type base language	optional

Clauses:

hint

Binding

The binding thread set for a **critical** region is all threads in the contention group.

Semantics

The *name* argument is used to identify the **critical** construct. For any **critical** construct for which *name* is not specified, the effect is as if an identical (unspecified) name was specified. The region that corresponds to a **critical** construct of a given name is executed as if only a single thread at a time among all threads in the contention group executes the region, without regard to the teams to which the threads belong.

C / C++

Identifiers used to identify a **critical** construct have external linkage and are in a name space that is separate from the name spaces used by labels, tags, members, and ordinary identifiers.

C / C++

Fortran

The names of **critical** constructs are global entities of the program. If a name conflicts with any other entity, the behavior of the program is unspecified.

Fortran

Execution Model Events

The *critical-acquiring* event occurs in a thread that encounters the **critical** construct on entry to the **critical** region before initiating synchronization for the region.

The *critical-acquired* event occurs in a thread that encounters the **critical** construct after it enters the region, but before it executes the structured block of the **critical** region.

The *critical-released* event occurs in a thread that encounters the **critical** construct after it completes any synchronization on exit from the **critical** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of a *critical-acquiring* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of a *critical-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_mutex_released** callback for each occurrence of a *critical-released* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

The callbacks occur in the task that encounters the critical construct. The callbacks should receive **ompt_mutex_critical** as their *kind* argument if practical, but a less specific kind is acceptable.

Restrictions

Restrictions to the **critical** construct are as follows:

- Unless the effect is as if **hint**(**omp_sync_hint_none**) was specified, the **critical** construct must specify a name.
- The *hint-expr* that is applied to each of the **critical** constructs with the same *name* must evaluate to the same value.

Fortran

- If a *name* is specified on a **critical** directive, the same *name* must also be specified on the **end critical** directive.
- If no *name* appears on the **critical** directive, no *name* can appear on the **end critical** directive.

Fortran

Cross References

- **ompt_callback_mutex_acquire_t**, see Section 19.5.2.14.
- **ompt_callback_mutex_t**, see Section 19.5.2.15.
- **ompt_mutex_critical**, see Section 19.4.4.17.
- Synchronization Hints, see Section 15.1.
- **hint** clause, see Section 15.1.2.

15.3 Barriers

15.3.1 barrier Construct

Name: barrier **Category:** executable	**Association:** none **Properties:** default

Binding

The binding thread set for a **barrier** region is the current team. A **barrier** region binds to the innermost enclosing **parallel** region.

Semantics

The **barrier** construct specifies an explicit barrier at the point at which the construct appears. Unless the binding region is canceled, all threads of the team that executes that binding region must enter the **barrier** region and complete execution of all explicit tasks bound to that binding region before any of the threads continue execution beyond the barrier.

The **barrier** region includes an implicit task scheduling point in the current task region.

Execution Model Events

The *explicit-barrier-begin* event occurs in each thread that encounters the **barrier** construct on entry to the **barrier** region.

The *explicit-barrier-wait-begin* event occurs when a task begins an interval of active or passive waiting in a **barrier** region.

The *explicit-barrier-wait-end* event occurs when a task ends an interval of active or passive waiting and resumes execution in a **barrier** region.

The *explicit-barrier-end* event occurs in each thread that encounters the **barrier** construct after the barrier synchronization on exit from the **barrier** region.

A *cancellation* event occurs if cancellation is activated at an implicit cancellation point in a **barrier** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_sync_region** callback with **ompt_sync_region_barrier_explicit** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *explicit-barrier-begin* event. Similarly, a thread dispatches a registered **ompt_callback_sync_region** callback with **ompt_sync_region_barrier_explicit** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of an *explicit-barrier-end* event. These callbacks occur in the context of the task that encountered the **barrier** construct and have type signature **ompt_callback_sync_region_t**.

A thread dispatches a registered **ompt_callback_sync_region_wait** callback with **ompt_sync_region_barrier_explicit** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *explicit-barrier-wait-begin* event. Similarly, a thread dispatches a registered **ompt_callback_sync_region_wait** callback with **ompt_sync_region_barrier_explicit** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of an *explicit-barrier-wait-end* event. These callbacks occur in the context of the task that encountered the **barrier** construct and have type signature **ompt_callback_sync_region_t**.

A thread dispatches a registered **ompt_callback_cancel** callback with **ompt_cancel_detected** as its *flags* argument for each occurrence of a *cancellation* event in that thread. The callback occurs in the context of the encountering task. The callback has type signature **ompt_callback_cancel_t**.

Restrictions

Restrictions to the **barrier** construct are as follows:

- Each **barrier** region must be encountered by all threads in a team or by none at all, unless cancellation has been requested for the innermost enclosing parallel region.
- The sequence of worksharing regions and **barrier** regions encountered must be the same for every thread in a team.

Cross References

- **ompt_callback_cancel_t**, see Section 19.5.2.18.
- **ompt_callback_sync_region_t**, see Section 19.5.2.13.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 19.4.4.11.
- **ompt_sync_region_barrier**, see Section 19.4.4.14.

15.3.2 Implicit Barriers

This section describes the OMPT events and tool callbacks associated with implicit barriers, which occur at the end of various regions as defined in the description of the constructs to which they correspond. Implicit barriers are task scheduling points. For a description of task scheduling points, associated events, and tool callbacks, see Section 12.9.

Execution Model Events

The *implicit-barrier-begin* event occurs in each implicit task at the beginning of an implicit barrier region.

The *implicit-barrier-wait-begin* event occurs when a task begins an interval of active or passive waiting in an implicit barrier region.

The *implicit-barrier-wait-end* event occurs when a task ends an interval of active or waiting and resumes execution of an implicit barrier region.

The *implicit-barrier-end* event occurs in each implicit task after the barrier synchronization on exit from an implicit barrier region.

A *cancellation* event occurs if cancellation is activated at an implicit cancellation point in an implicit barrier region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_sync_region** callback for each implicit barrier *begin* and *end* event. Similarly, a thread dispatches a registered **ompt_callback_sync_region_wait** callback for each implicit barrier *wait-begin* and *wait-end* event. All callbacks for implicit barrier events execute in the context of the encountering task and have type signature **ompt_callback_sync_region_t**.

For the implicit barrier at the end of a worksharing construct, the *kind* argument is **ompt_sync_region_barrier_implicit_workshare**. For the implicit barrier at the end of a **parallel** region, the *kind* argument is **ompt_sync_region_barrier_implicit_parallel**. For an extra barrier added by an OpenMP implementation, the *kind* argument is **ompt_sync_region_barrier_implementation**. For a barrier at the end of a **teams** region, the *kind* argument is **ompt_sync_region_barrier_teams**.

A thread dispatches a registered **ompt_callback_cancel** callback with **ompt_cancel_detected** as its *flags* argument for each occurrence of a *cancellation* event in that thread. The callback occurs in the context of the encountering task. The callback has type signature **ompt_callback_cancel_t**.

Restrictions

Restrictions to implicit barriers are as follows:

- If a thread is in the state **ompt_state_wait_barrier_implicit_parallel**, a call to **ompt_get_parallel_info** may return a pointer to a copy of the data object associated with the parallel region rather than a pointer to the associated data object itself. Writing to the data object returned by **omp_get_parallel_info** when a thread is in the **ompt_state_wait_barrier_implicit_parallel** results in unspecified behavior.

Cross References

- **ompt_callback_cancel_t**, see Section 19.5.2.18.
- **ompt_callback_sync_region_t**, see Section 19.5.2.13.
- **ompt_cancel_detected**, see Section 19.4.4.26.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 19.4.4.11.
- **ompt_sync_region_barrier_implementation**, **ompt_sync_region_barrier_implicit_parallel** **ompt_sync_region_barrier_teams**, and **ompt_sync_region_barrier_implicit_workshare**, see Section 19.4.4.14.

15.3.3 Implementation-Specific Barriers

An OpenMP implementation can execute implementation-specific barriers that the OpenMP specification does not imply; therefore, no *execution model events* are bound to them. The implementation can handle these barriers like implicit barriers and dispatch all events as for implicit barriers. These callbacks use **ompt_sync_region_barrier_implementation** — or **ompt_sync_region_barrier**, if the implementation cannot make a distinction — as the *kind* argument when they are dispatched.

15.4 taskgroup Construct

Name: taskgroup **Category:** executable	**Association:** block **Properties:** cancellable

Clauses:

allocate, **task_reduction**

Binding

The binding task set of a **taskgroup** region is all tasks of the current team that are generated in the region. A **taskgroup** region binds to the innermost enclosing **parallel** region.

Semantics

The **taskgroup** construct specifies a wait on completion of child tasks of the current task and their descendent tasks. When a thread encounters a **taskgroup** construct, it starts executing the region. All child tasks generated in the **taskgroup** region and all of their descendants that bind to the same **parallel** region as the **taskgroup** region are part of the *taskgroup set* associated with the **taskgroup** region.

An implicit task scheduling point occurs at the end of the **taskgroup** region. The current task is suspended at the task scheduling point until all tasks in the *taskgroup set* complete execution.

Execution Model Events

The *taskgroup-begin* event occurs in each thread that encounters the **taskgroup** construct on entry to the **taskgroup** region.

The *taskgroup-wait-begin* event occurs when a task begins an interval of active or passive waiting in a **taskgroup** region.

The *taskgroup-wait-end* event occurs when a task ends an interval of active or passive waiting and resumes execution in a **taskgroup** region.

The *taskgroup-end* event occurs in each thread that encounters the **taskgroup** construct after the taskgroup synchronization on exit from the **taskgroup** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_sync_region** callback with **ompt_sync_region_taskgroup** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *taskgroup-begin* event in the task that encounters the **taskgroup** construct. Similarly, a thread dispatches a registered **ompt_callback_sync_region** callback with **ompt_sync_region_taskgroup** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of a *taskgroup-end* event in the task that encounters the **taskgroup** construct. These callbacks occur in the task that encounters the **taskgroup** construct and have the type signature **ompt_callback_sync_region_t**.

A thread dispatches a registered **ompt_callback_sync_region_wait** callback with **ompt_sync_region_taskgroup** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *taskgroup-wait-begin* event. Similarly, a thread dispatches a registered **ompt_callback_sync_region_wait** callback with **ompt_sync_region_taskgroup** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of a *taskgroup-wait-end* event. These callbacks occur in the context of the task that encounters the **taskgroup** construct and have type signature **ompt_callback_sync_region_t**.

Cross References

- **ompt_callback_sync_region_t**, see Section 19.5.2.13.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 19.4.4.11.
- **ompt_sync_region_taskgroup**, see Section 19.4.4.14.
- Task scheduling, see Section 12.9.
- **task_reduction** clause, see Section 5.5.10.

15.5 taskwait Construct

Name: **taskwait** **Category:** executable	**Association:** none **Properties:** default

Clauses:
depend, **nowait**

Binding

The **taskwait** region binds to the current task region. The binding thread set of the **taskwait** region is the current team.

Semantics

The **taskwait** construct specifies a wait on the completion of child tasks of the current task.

If no **depend** clause is present on the **taskwait** construct, the current task region is suspended at an implicit task scheduling point associated with the construct. The current task region remains suspended until all child tasks that it generated before the **taskwait** region complete execution.

If one or more **depend** clauses are present on the **taskwait** construct and the **nowait** clause is not also present, the behavior is as if these clauses were applied to a **task** construct with an empty associated structured block that generates a *mergeable* and *included task*. Thus, the current task region is suspended until the *predecessor tasks* of this task complete execution.

If one or more **depend** clauses are present on the **taskwait** construct and the **nowait** clause is also present, the behavior is as if these clauses were applied to a **task** construct with an empty associated structured block that generates a task for which execution may be deferred. Thus, all *predecessor tasks* of this task must complete execution before any subsequently generated task that depends on this task starts its execution.

Execution Model Events

The *taskwait-begin* event occurs in a thread when it encounters a **taskwait** construct with no **depend** clause on entry to the **taskwait** region.

The *taskwait-wait-begin* event occurs when a task begins an interval of active or passive waiting in a region corresponding to a **taskwait** construct with no **depend** clause.

The *taskwait-wait-end* event occurs when a task ends an interval of active or passive waiting and resumes execution from a region corresponding to a **taskwait** construct with no **depend** clause.

The *taskwait-end* event occurs in a thread when it encounters a **taskwait** construct with no **depend** clause after the taskwait synchronization on exit from the **taskwait** region.

The *taskwait-init* event occurs in a thread when it encounters a **taskwait** construct with one or more **depend** clauses on entry to the **taskwait** region.

The *taskwait-complete* event occurs on completion of the dependent task that results from a **taskwait** construct with one or more **depend** clauses, in the context of the thread that executes the dependent task and before any subsequently generated task that depends on the dependent task starts its execution.

Tool Callbacks

A thread dispatches a registered **ompt_callback_sync_region** callback with **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *taskwait-begin* event in the task that encounters the **taskwait** construct. Similarly, a thread dispatches a registered **ompt_callback_sync_region** callback with **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of a *taskwait-end* event in the task that encounters the **taskwait** construct. These callbacks occur in the task that encounters the **taskwait** construct and have the type signature **ompt_callback_sync_region_t**.

A thread dispatches a registered **ompt_callback_sync_region_wait** callback with **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *taskwait-wait-begin* event. Similarly, a thread dispatches a registered **ompt_callback_sync_region_wait** callback with **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of a *taskwait-wait-end* event. These callbacks occur in the context of the task that encounters the **taskwait** construct and have type signature **ompt_callback_sync_region_t**.

A thread dispatches a registered **ompt_callback_task_create** callback for each occurrence of a *taskwait-init* event in the context of the encountering task. This callback has the type signature **ompt_callback_task_create_t**. In the dispatched callback, **(***flags* **& ompt_task_taskwait)** always evaluates to *true*. If the **nowait** clause is not present, **(***flags* **& ompt_task_undeferred)** also evaluates to true.

A thread dispatches a registered **ompt_callback_task_schedule** callback for each occurrence of a *taskwait-complete* event. This callback has the type signature **ompt_callback_task_schedule_t** with **ompt_taskwait_complete** as its *prior_task_status* argument.

Restrictions

Restrictions to the **taskwait** construct are as follows:

- The **mutexinoutset** *dependence-type* may not appear in a **depend** clause on a **taskwait** construct.
- If the *dependence-type* of a **depend** clause is **depobj** then the dependence objects cannot represent dependences of the **mutexinoutset** dependence type.
- The **nowait** clause may only appear on a **taskwait** directive if the **depend** clause is present.

Cross References

- **ompt_callback_sync_region_t**, see Section 19.5.2.13.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 19.4.4.11.
- **ompt_sync_region_taskwait**, see Section 19.4.4.14.
- Task scheduling, see Section 12.9.
- **depend** clause, see Section 15.9.5.
- **nowait** clause, see Section 15.6.
- **task** construct, see Section 12.5.

15.6 nowait Clause

Name:	**Properties:**
nowait	unique, end-clause

Directives:

dispatch, **do**, **for**, **interop**, **scope**, **sections**, **single**, **target**, **target enter data**, **target exit data**, **target update**, **taskwait**, **workshare**

Semantics

The **nowait** clause overrides any synchronization that would otherwise occur at the end of a construct. It can also specify that an *interoprability requirement set* includes the *nowait* propoerty.

If the construct includes an implicit barrier, the **nowait** clause specifies that the barrier will not occur.

For constructs that generate a task, the **nowait** clause specifies that the generated task may be deferred. If the **nowait** clause is not present on the directive then the generated task is an included task (so it executes synchronously in the context of the encountering task).

For constructs that generate an *interoprability requirement set*, the **nowait** clause adds the *nowait* property to the set.

Cross References

- **scope** construct, see Section 11.2.
- Worksharing-loop construct, see Section 11.5.
- **dispatch** construct, see Section 7.6.
- **interop** construct, see Section 14.1.
- **sections** construct, see Section 11.3.
- **single** construct, see Section 11.1.
- **target** construct, see Section 13.8.
- **target enter data** construct, see Section 13.6.
- **target exit data** construct, see Section 13.7.
- **target update** construct, see Section 13.9.
- **taskwait** construct, see Section 15.5.
- **workshare** construct, see Section 11.4.

15.7 nogroup Clause

Name:	**Properties:**
nogroup	unique, end-clause

Directives:
taskloop

Semantics

The **nogroup** clause overrides any implicit **taskgroup** that would otherwise occur at the end of a construct.

Cross References

- **taskloop** construct, see Section 12.6.

15.8 OpenMP Memory Ordering

This sections describes constructs and clauses in OpenMP that support ordering of memory operations.

15.8.1 *memory-order* Clauses

Clause group: memory-order

Properties: unique, inarguable, fully exclusive	**Members:** **`acq_rel`**, **`acquire`**, **`relaxed`**, **`release`**, **`seq_cst`**

Semantics

The *memory-order* clause grouping defines a set of clauses that indicate the memory ordering requirements for the visibility of the effects of the constructs on which they may be specified.

Cross References

- **`requires`** directive, see Section 8.2.
- **`atomic`** construct, see Section 15.8.4.
- **`flush`** construct, see Section 15.8.5.

15.8.2 *atomic* Clauses

Clause group: atomic

Properties: unique, inarguable, fully exclusive	**Members:** **`read`**, **`update`**, **`write`**

Semantics

The *atomic* clause grouping defines a set of clauses that defines the semantics for which a directive enforces atomicity. For constructs that accept the *atomic* clause grouping, the effect is as if the **`update`** clause is specified if no member of the grouping is specified.

Cross References

- **`atomic`** construct, see Section 15.8.4.

15.8.3 *extended-atomic* Clauses

Clause group: extended-atomic

Properties: unique	**Members:** **`capture`**, **`compare`**, **`fail`**, **`weak`**

Semantics

The *extended-atomic* clause grouping defines a set of clauses that extend the atomicity semantics specified by members of the *atomic* clause grouping. The **capture** clause extends the semantics to capture the value of the variable being updated atomically. The **compare** clause extends the semantics to perform the atomic update conditionally.

The **weak** clause specifies that the comparison performed by a conditional atomic update may spuriously fail, evaluating to not equal even when the values are equal.

Note – Allowing for spurious failure by specifying a **weak** clause can result in performance gains on some systems when using compare-and-swap in a loop. For cases where a single compare-and-swap would otherwise be sufficient, using a loop over a **weak** compare-and-swap is unlikely to improve performance.

The **fail** clause extends the semantics to specify the memory ordering requirements for any comparison performed by any atomic conditional update that fails. Its argument overrides any other specified memory ordering.

Restrictions

Restrictions to the **atomic** construct are as follows:

- **acq_rel** and **release** cannot be specified as arguments to the **fail** clause.

15.8.4 atomic Construct

Name: atomic **Category:** executable	**Association:** block (atomic structured block) **Properties:** simdizable

Clause groups:
atomic, **extended-atomic**, **memory-order**

Clauses:
hint

Binding

If the size of *x* is 8, 16, 32, or 64 bits and *x* is aligned to a multiple of its size, the binding thread set for the **atomic** region is all threads on the device. Otherwise, the binding thread set for the **atomic** region is all threads in the contention group. **atomic** regions enforce exclusive access with respect to other **atomic** regions that access the same storage location *x* among all threads in the binding thread set without regard to the teams to which the threads belong.

Semantics

The **atomic** construct ensures that a specific storage location is accessed atomically so that possible simultaneous reads and writes by multiple threads do not result in indeterminate values. The **atomic** construct with the **read** clause results in an atomic read of the location designated by *x*. The **atomic** construct with the **write** clause results in an atomic write of the location designated by *x*. The **atomic** construct with the **update** clause results in an atomic update of the location designated by *x* using the designated operator or intrinsic. Only the read and write of the location designated by *x* are performed mutually atomically. The evaluation of *expr* or *expr-list* need not be atomic with respect to the read or write of the location designated by *x*. No task scheduling points are allowed between the read and the write of the location designated by *x*.

If the **capture** clause is present, the atomic update is an atomic captured update — an atomic update to the location designated by *x* using the designated operator or intrinsic while also capturing the original or final value of the location designated by *x* with respect to the atomic update. The original or final value of the location designated by *x* is written in the location designated by *v* based on the base language semantics of structured block or statements of the **atomic** construct. Only the read and write of the location designated by *x* are performed mutually atomically. Neither the evaluation of *expr* or *expr-list*, nor the write to the location designated by *v*, need be atomic with respect to the read or write of the location designated by *x*.

If the **compare** clause is present, the atomic update is an atomic conditional update. For forms that use an equality comparison, the operation is an atomic compare-and-swap. It atomically compares the value of *x* to *e* and writes the value of *d* into the location designated by *x* if they are equal. Based on the base language semantics of the associated structured block, the original or final value of the location designated by *x* is written to the location designated by *v*, which is allowed to be the same location as designated by *e*, or the result of the comparison is written to the location designated by *r*. Only the read and write of the location designated by *x* are performed mutually atomically. Neither the evaluation of either *e* or *d* nor writes to the locations designated by *v* and *r* need be atomic with respect to the read or write of the location designated by *x*.

C / C++

If the **compare** clause is present, forms that use *ordop* are logically an atomic maximum or minimum, but they may be implemented with a compare-and-swap loop with short-circuiting. For forms where *statement* is *cond-expr-stmt*, if the result of the condition implies that the value of *x* does not change then the update may not occur.

C / C++

If a *memory-order* clause is present, or implicitly provided by a **requires** directive, it specifies the effective memory ordering. Otherwise the effect is as if the **relaxed** memory ordering clause is specified.

The **atomic** construct may be used to enforce memory consistency between threads, based on the guarantees provided by Section 1.4.6. A strong flush on the location designated by *x* is performed on entry to and exit from the atomic operation, ensuring that the set of all atomic operations applied to the same location in a race-free program has a total completion order. If the **write** or **update** clause is specified, the atomic operation is not an atomic conditional update for which the comparison fails, and the effective memory ordering is **release**, **acq_rel**, or **seq_cst**, the strong flush on entry to the atomic operation is also a release flush. If the **read** or **update** clause is specified and the effective memory ordering is **acquire**, **acq_rel**, or **seq_cst** then the strong flush on exit from the atomic operation is also an acquire flush. Therefore, if the effective memory ordering is not **relaxed**, release and/or acquire flush operations are implied and permit synchronization between the threads without the use of explicit **flush** directives.

For all forms of the **atomic** construct, any combination of two or more of these **atomic** constructs enforces mutually exclusive access to the locations designated by *x* among threads in the binding thread set. To avoid data races, all accesses of the locations designated by *x* that could potentially occur in parallel must be protected with an **atomic** construct.

atomic regions do not guarantee exclusive access with respect to any accesses outside of **atomic** regions to the same storage location *x* even if those accesses occur during a **critical** or **ordered** region, while an OpenMP lock is owned by the executing task, or during the execution of a **reduction** clause.

However, other OpenMP synchronization can ensure the desired exclusive access. For example, a barrier that follows a series of atomic updates to *x* guarantees that subsequent accesses do not form a race with the atomic accesses.

A compliant implementation may enforce exclusive access between **atomic** regions that update different storage locations. The circumstances under which this occurs are implementation defined.

If the storage location designated by *x* is not size-aligned (that is, if the byte alignment of *x* is not a multiple of the size of *x*), then the behavior of the **atomic** region is implementation defined.

Execution Model Events

The *atomic-acquiring* event occurs in the thread that encounters the **atomic** construct on entry to the atomic region before initiating synchronization for the region.

The *atomic-acquired* event occurs in the thread that encounters the **atomic** construct after it enters the region, but before it executes the structured block of the **atomic** region.

The *atomic-released* event occurs in the thread that encounters the **atomic** construct after it completes any synchronization on exit from the **atomic** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of an *atomic-acquiring* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of an *atomic-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_mutex_released** callback with **ompt_mutex_atomic** as the *kind* argument if practical, although a less specific *kind* may be used, for each occurrence of an *atomic-released* event in that thread. This callback has the type signature **ompt_callback_mutex_t** and occurs in the task that encounters the atomic construct.

Restrictions

Restrictions to the **atomic** construct are as follows:

- OpenMP constructs may not be encountered during execution of an **atomic** region.
- If a **capture** or **compare** clause is specified, the *atomic* clause must be **update**.
- If a **capture** clause is specified but the **compare** clause is not specifed, an *update-capture-atomic* structured block must be associated with the construct.
- If both **capture** and **compare** clauses are specified, a *conditional-update-capture-atomic* structured block must be associated with the construct.
- If a **compare** clause is specified but the **capture** clause is not specifed, a *conditional-update-atomic* structured block must be associated with the construct.
- If a **write** clause is specified, a *write-atomic* structured block must be associated with the construct.
- If a **read** clause is specified, a *read-atomic* structured block must be associated with the construct.
- If the *atomic* clause is **read** then the *memory-order* clause must not be **release**.
- If the *atomic* clause is **write** then the *memory-order* clause must not be **acquire**.
- The **weak** clause may only appear if the resulting atomic operation is an atomic conditional update for which the comparison tests for equality.

C / C++

- All atomic accesses to the storage locations designated by *x* throughout the program are required to have a compatible type.
- The **fail** clause may only appear if the resulting atomic operation is an atomic conditional update.

C / C++

Fortran

- All atomic accesses to the storage locations designated by *x* throughout the program are required to have the same type and type parameters.
- The **fail** clause may only appear if the resulting atomic operation is an atomic conditional update or an atomic update where *intrinsic-procedure-name* is either **MAX** or **MIN**.

Fortran

Cross References

- lock routines, see Section 18.9.
- *memory-order* clauses, see Section 15.8.1.
- **`ompt_callback_mutex_acquire_t`**, see Section 19.5.2.14.
- **`ompt_callback_mutex_t`**, see Section 19.5.2.15.
- **`ompt_mutex_atomic`**, see Section 19.4.4.17.
- OpenMP atomic structured blocks, see Section 4.3.3.
- Synchronization hints, see Section 15.1.
- **`barrier`** construct, see Section 15.3.1.
- **`critical`** construct, see Section 15.2.
- **`flush`** construct, see Section 15.8.5.
- **`hint`** clause, see Section 15.1.2.
- **`ordered`** construct, see Section 15.9.7.
- **`requires`** directive, see Section 8.2.
- **`reduction`** clause, see Section 5.5.9.

15.8.5 `flush` Construct

Name: `flush` **Category:** executable	**Association:** none **Properties:** default

Arguments: `flush` (*list*)

Name	Type	Properties
list	List containing variable list item	optional

Clause groups:

`memory-order`

Binding

The binding thread set for a **`flush`** region is all threads in the *device-set* of its flush operation.

Semantics

The **`flush`** construct executes the OpenMP flush operation. This operation makes a thread's temporary view of memory consistent with memory and enforces an order on the memory operations of the variables explicitly specified or implied. Execution of a **`flush`** region affects the memory and it affects the temporary view of memory of the encountering thread. It does not affect the temporary view of other threads. Other threads on devices in the *device-set* must themselves execute a flush operation in order to be guaranteed to observe the effects of the flush operation of the encountering thread. See the memory model description in Section 1.4 for more details.

If neither a *memory-order* clause nor a *list* argument appears on a **`flush`** construct then the behavior is as if the *memory-order* clause is **`seq_cst`**.

A **`flush`** construct with the **`seq_cst`** clause, executed on a given thread, operates as if all data storage blocks that are accessible to the thread are flushed by a strong flush operation. A **`flush`** construct with a list applies a strong flush operation to the items in the list, and the flush operation does not complete until the operation is complete for all specified list items. An implementation may implement a **`flush`** construct with a list by ignoring the list and treating it the same as a **`flush`** construct with the **`seq_cst`** clause.

If no list items are specified, the flush operation has the release and/or acquire flush properties:

- If the *memory-order* clause is **`seq_cst`** or **`acq_rel`**, the flush operation is both a release flush and an acquire flush.
- If the *memory-order* clause is **`release`**, the flush operation is a release flush.
- If the *memory-order* clause is **`acquire`**, the flush operation is an acquire flush.

C / C++

If a pointer is present in the list, the pointer itself is flushed, not the memory block to which the pointer refers.

A **`flush`** construct without a list corresponds to a call to **`atomic_thread_fence`**, where the argument is given by the identifier that results from prefixing **`memory_order_`** to the *memory-order* clause name.

For a **`flush`** construct without a list, the generated **`flush`** region implicitly performs the corresponding call to **`atomic_thread_fence`**. The behavior of an explicit call to **`atomic_thread_fence`** that occurs in the program and does not have the argument **`memory_order_consume`** is as if the call is replaced by its corresponding **`flush`** construct.

C / C++

Fortran

If the list item or a subobject of the list item has the **POINTER** attribute, the allocation or association status of the **POINTER** item is flushed, but the pointer target is not. If the list item is a Cray pointer, the pointer is flushed, but the object to which it points is not. Cray pointer support has been deprecated. If the list item is of type **C_PTR**, the variable is flushed, but the storage that corresponds to that address is not flushed. If the list item or the subobject of the list item has the **ALLOCATABLE** attribute and has an allocation status of allocated, the allocated variable is flushed; otherwise the allocation status is flushed.

Fortran

Execution Model Events

The *flush* event occurs in a thread that encounters the **flush** construct.

Tool Callbacks

A thread dispatches a registered **ompt_callback_flush** callback for each occurrence of a *flush* event in that thread. This callback has the type signature **ompt_callback_flush_t**.

Restrictions

Restrictions to the **flush** construct are as follows:

- If a *memory-order* clause is specified, the *list* argument must not be specified.
- The *memory-order* clause must not be **relaxed**.

Cross References

- *memory-order* clauses, see Section 15.8.1.
- **ompt_callback_flush_t**, see Section 19.5.2.17.

15.8.6 Implicit Flushes

Flush operations implied when executing an **atomic** region are described in Section 15.8.4.

A **flush** region that corresponds to a **flush** directive with the **release** clause present is implied at the following locations:

- During a barrier region;
- At entry to a **parallel** region;
- At entry to a **teams** region;
- At exit from a **critical** region;
- During an **omp_unset_lock** region;
- During an **omp_unset_nest_lock** region;

- During an **omp_fulfill_event** region;
- Immediately before every task scheduling point;
- At exit from the task region of each implicit task;
- At exit from an **ordered** region, if a **threads** clause or a **depend** clause with a **source** dependence type is present, or if no clauses are present; and
- During a **cancel** region, if the *cancel-var* ICV is *true*.

For a **target** construct, the *device-set* of an implicit release flush that is performed in a target task during the generation of the **target** region and that is performed on exit from the initial task region that implicitly encloses the **target** region consists of the devices that execute the target task and the **target** region.

A **flush** region that corresponds to a **flush** directive with the **acquire** clause present is implied at the following locations:

- During a barrier region;
- At exit from a **teams** region;
- At entry to a **critical** region;
- If the region causes the lock to be set, during:
  - an **omp_set_lock** region;
  - an **omp_test_lock** region;
  - an **omp_set_nest_lock** region; and
  - an **omp_test_nest_lock** region;
- Immediately after every task scheduling point;
- At entry to the task region of each implicit task;
- At entry to an **ordered** region, if a **threads** clause or a **depend** clause with a **sink** dependence type is present, or if no clauses are present; and
- Immediately before a cancellation point, if the *cancel-var* ICV is *true* and cancellation has been activated.

For a **target** construct, the *device-set* of an implicit acquire flush that is performed in a target task following the generation of the **target** region or that is performed on entry to the initial task region that implicitly encloses the **target** region consists of the devices that execute the target task and the **target** region.

Note – A **flush** region is not implied at the following locations:

- At entry to worksharing regions; and
- At entry to or exit from **masked** regions.

The synchronization behavior of implicit flushes is as follows:

- When a thread executes an **atomic** region for which the corresponding construct has the **release**, **acq_rel**, or **seq_cst** clause and specifies an atomic operation that starts a given release sequence, the release flush that is performed on entry to the atomic operation synchronizes with an acquire flush that is performed by a different thread and has an associated atomic operation that reads a value written by a modification in the release sequence.
- When a thread executes an **atomic** region for which the corresponding construct has the **acquire**, **acq_rel**, or **seq_cst** clause and specifies an atomic operation that reads a value written by a given modification, a release flush that is performed by a different thread and has an associated release sequence that contains that modification synchronizes with the acquire flush that is performed on exit from the atomic operation.
- When a thread executes a **critical** region that has a given name, the behavior is as if the release flush performed on exit from the region synchronizes with the acquire flush performed on entry to the next **critical** region with the same name that is performed by a different thread, if it exists.
- When a thread team executes a **barrier** region, the behavior is as if the release flush performed by each thread within the region, and the release flush performed by any other thread upon fulfilling the *allow-completion* event for a detachable task bound to the binding parallel region of the region, synchronizes with the acquire flush performed by all other threads within the region.
- When a thread executes a **taskwait** region that does not result in the creation of a dependent task and the task that encounters the corresponding **taskwait** construct has at least one child task, the behavior is as if each thread that executes a child task that is generated before the **taskwait** region performs a release flush upon completion of the associated structured block of the child task that synchronizes with an acquire flush performed in the **taskwait** region. If the child task is detachable, the thread that fulfills its *allow-completion* event performs a release flush upon fulfilling the event that synchronizes with the acquire flush performed in the **taskwait** region.
- When a thread executes a **taskgroup** region, the behavior is as if each thread that executes a remaining descendant task performs a release flush upon completion of the associated structured block of the descendant task that synchronizes with an acquire flush performed on exit from the **taskgroup** region. If the descendant task is detachable, the thread that fulfills its *allow-completion* event performs a release flush upon fulfilling the event that synchronizes with the acquire flush performed in the **taskgroup** region.

- When a thread executes an **ordered** region that does not arise from a stand-alone **ordered** directive, the behavior is as if the release flush performed on exit from the region synchronizes with the acquire flush performed on entry to an **ordered** region encountered in the next logical iteration to be executed by a different thread, if it exists.
- When a thread executes an **ordered** region that arises from a stand-alone **ordered** directive, the behavior is as if the release flush performed in the **ordered** region from a given source iteration synchronizes with the acquire flush performed in all **ordered** regions executed by a different thread that are waiting for dependences on that iteration to be satisfied.
- When a thread team begins execution of a **parallel** region, the behavior is as if the release flush performed by the primary thread on entry to the **parallel** region synchronizes with the acquire flush performed on entry to each implicit task that is assigned to a different thread.
- When an initial thread begins execution of a **target** region that is generated by a different thread from a target task, the behavior is as if the release flush performed by the generating thread in the target task synchronizes with the acquire flush performed by the initial thread on entry to its initial task region.
- When an initial thread completes execution of a **target** region that is generated by a different thread from a target task, the behavior is as if the release flush performed by the initial thread on exit from its initial task region synchronizes with the acquire flush performed by the generating thread in the target task.
- When a thread encounters a **teams** construct, the behavior is as if the release flush performed by the thread on entry to the **teams** region synchronizes with the acquire flush performed on entry to each initial task that is executed by a different initial thread that participates in the execution of the **teams** region.
- When a thread that encounters a **teams** construct reaches the end of the **teams** region, the behavior is as if the release flush performed by each different participating initial thread at exit from its initial task synchronizes with the acquire flush performed by the thread at exit from the **teams** region.
- When a task generates an explicit task that begins execution on a different thread, the behavior is as if the thread that is executing the generating task performs a release flush that synchronizes with the acquire flush performed by the thread that begins to execute the explicit task.
- When an undeferred task completes execution on a given thread that is different from the thread on which its generating task is suspended, the behavior is as if a release flush performed by the thread that completes execution of the associated structured block of the undeferred task synchronizes with an acquire flush performed by the thread that resumes execution of the generating task.
- When a dependent task with one or more predecessor tasks begins execution on a given thread, the behavior is as if each release flush performed by a different thread on completion of the associated structured block of a predecessor task synchronizes with the acquire flush performed by the thread that begins to execute the dependent task. If the predecessor task is detachable, the

thread that fulfills its *allow-completion* event performs a release flush upon fulfilling the event that synchronizes with the acquire flush performed when the dependent task begins to execute.

- When a task begins execution on a given thread and it is mutually exclusive with respect to another sibling task that is executed by a different thread, the behavior is as if each release flush performed on completion of the sibling task synchronizes with the acquire flush performed by the thread that begins to execute the task.
- When a thread executes a **cancel** region, the *cancel-var* ICV is *true*, and cancellation is not already activated for the specified region, the behavior is as if the release flush performed during the **cancel** region synchronizes with the acquire flush performed by a different thread immediately before a cancellation point in which that thread observes cancellation was activated for the region.
- When a thread executes an **omp_unset_lock** region that causes the specified lock to be unset, the behavior is as if a release flush is performed during the **omp_unset_lock** region that synchronizes with an acquire flush that is performed during the next **omp_set_lock** or **omp_test_lock** region to be executed by a different thread that causes the specified lock to be set.
- When a thread executes an **omp_unset_nest_lock** region that causes the specified nested lock to be unset, the behavior is as if a release flush is performed during the **omp_unset_nest_lock** region that synchronizes with an acquire flush that is performed during the next **omp_set_nest_lock** or **omp_test_nest_lock** region to be executed by a different thread that causes the specified nested lock to be set.

15.9 OpenMP Dependences

This sections describes constructs and clauses in OpenMP that support the specification and enforcement of dependences. OpenMP supports two kinds of dependences: *task dependences*, which enforce orderings between tasks; and *cross-iteration dependences*, which enforce orderings between loop iterations.

15.9.1 *task-dependence-type* Modifiers

Modifiers:

Name	Modifies	Type	Properties
task-dependence-type	*locator-list*	**Keyword: depobj**, **in**, **inout**, **inoutset**, **mutexinoutset**, **out**	ultimate, unique

Semantics

OpenMP clauses that are related to task dependences use the *task-dependence-type* modifier to identify the type of dependence relevant to that clause. The effect of the type of dependence is associated with locator list items as described with the **depend** clause, see Section 15.9.5.

Cross References

- **depend** clause, see Section 15.9.5.
- **depobj** construct, see Section 15.9.4.
- **update** clause, see Section 15.9.3.

15.9.2 Depend Objects

OpenMP depend objects can be used to supply user-computed dependences to **depend** clauses. OpenMP depend objects must be accessed only through the **depobj** construct or through the **depend** clause; programs that otherwise access OpenMP depend objects are non-conforming.

An OpenMP depend object can be in one of the following states: *uninitialized* or *initialized*. Initially OpenMP depend objects are in the *uninitialized* state.

15.9.3 update Clause

Name:	Properties:
update-depend_objects	unique

Arguments:

Name	Type	Properties
task-dependence-type	reference	default

Directives:

depobj

Semantics

The **update** clause sets the dependence type of an OpenMP depend object to *task-dependence-type*.

Restrictions

Restrictions to the **update** clause are as follows:

- *task-dependence-type* must not be **depobj**.

Cross References

- **depobj** construct, see Section 15.9.4.
- *task-dependence-modifiers* construct, see Section 15.9.1.

15.9.4 depobj Construct

Name: depobj **Category:** executable	**Association:** none **Properties:** default

Arguments: depobj (*depend_object*)

Name	Type	Properties
depend_object	Variable of type depend type	default

Clauses:
depend, **destroy**, **update**

Clause set:

Properties: fully exclusive, required	**Members:** depend, destroy, update

Binding

The binding thread set for a **depobj** region is the encountering thread.

Semantics

The **depobj** construct initializes, updates or destroys an OpenMP depend object. If a **depend** clause is specified, the state of *depend_object* is set to initialized and *depend_object* is set to represent the dependence that the **depend** clause specifies. If an **update** clause is specified, *depend_object* is updated to represent the new *task-dependence-type*. If a **destroy** clause is specified, the state of *depend_object* is set to uninitialized.

Restrictions

Restrictions to the **depobj** construct are as follows:

- An **update** clause on a **depobj** construct must not specify the **depobj** *task-dependence-type*.
- A **depend** clause on a **depobj** construct can only specify one locator.
- *depend_object* must be in the uninitialized state if a **depend** clause is specified.
- *depend_object* must be in the initialized state if a **destroy** clause is specified.
- *depend_object* must be in the initialized state if a **update** clause is specified.

Cross References

- **destroy** clause, see Section 3.5.
- **depend** clause, see Section 15.9.5.
- *task-dependence-modifiers* construct, see Section 15.9.1.
- **update** clause, see Section 15.9.3.

15.9.5 `depend` Clause

Name:	**Properties:**
`depend`	default

Arguments:

Name	Type	Properties
locator-list	List containing locator list item	default

Modifiers:

Name	Modifies	Type	Properties
task-dependence-type	*locator-list*	reference	default
depend-modifier	*locator-list*	iterator modifier	unique

Directives:

`depobj`, **`interop`**, **`target`**, **`target enter data`**, **`target exit data`**, **`target update`**, **`task`**, **`taskwait`**

Semantics

The **`depend`** clause enforces additional constraints on the scheduling of taskss. These constraints establish dependences only between sibling tasks. Task dependences are derived from the *task-dependence-type* and the list items.

The storage location of a list item matches the storage location of another list item if they have the same storage location, or if any of the list items is **`omp_all_memory`**.

For the **`in`** *task-dependence-type*, if the storage location of at least one of the list items matches the storage location of a list item appearing in a **`depend`** clause with an **`out`**, **`inout`**, **`mutexinoutset`**, or **`inoutset`** *task-dependence-type* on a construct from which a sibling task was previously generated, then the generated task will be a dependent task of that sibling task.

For the **`out`** and **`inout`** *task-dependence-types*, if the storage location of at least one of the list items matches the storage location of a list item appearing in a **`depend`** clause with an **`in`**, **`out`**, **`inout`**, **`mutexinoutset`**, or **`inoutset`** *task-dependence-type* on a construct from which a sibling task was previously generated, then the generated task will be a dependent task of that sibling task.

For the **`mutexinoutset`** *task-dependence-type*, if the storage location of at least one of the list items matches the storage location of a list item appearing in a **`depend`** clause with an **`in`**, **`out`**, **`inout`**, or **`inoutset`** *task-dependence-type* on a construct from which a sibling task was previously generated, then the generated task will be a dependent task of that sibling task.

If a list item appearing in a **`depend`** clause with a **`mutexinoutset`** *task-dependence-type* on a task generating construct matches a list item appearing in a **`depend`** clause with a **`mutexinoutset`** *task-dependence-type* on a different task generating construct, and both constructs generate sibling tasks, the sibling tasks will be mutually exclusive tasks.

For the **inoutset** *task-dependence-type*, if the storage location of at least one of the list items matches the storage location of a list item appearing in a **depend** clause with an **in**, **out**, **inout**, or **mutexinoutset** *task-dependence-type* on a construct from which a sibling task was previously generated, then the generated task will be a dependent task of that sibling task.

When the *task-dependence-type* is **depobj**, the task dependences are derived from the dependences represented by the depend objects specified in the **depend** clause as if the **depend** clauses of the **depobj** constructs were specified in the current construct.

The list items that appear in the **depend** clause may reference iterators defined by an *iterators-definition* appearing on an **iterator** modifier.

The list items that appear in the **depend** clause may include array sections or the **omp_all_memory** reserved locator.

Fortran

If a list item has the **ALLOCATABLE** attribute and its allocation status is unallocated, the behavior is unspecified. If a list item has the **POINTER** attribute and its association status is disassociated or undefined, the behavior is unspecified.

Fortran

C / C++

The list items that appear in a **depend** clause may use shape-operators.

C / C++

Note – The enforced task dependence establishes a synchronization of memory accesses performed by a dependent task with respect to accesses performed by the predecessor tasks. However, the programmer must properly synchronize with respect to other concurrent accesses that occur outside of those tasks.

Execution Model Events

The *task-dependences* event occurs in a thread that encounters a task generating construct or a **taskwait** construct with a **depend** clause immediately after the *task-create* event for the new task or the *taskwait-init* event.

The *task-dependence* event indicates an unfulfilled dependence for the generated task. This event occurs in a thread that observes the unfulfilled dependence before it is satisfied.

Tool Callbacks

A thread dispatches the **ompt_callback_dependences** callback for each occurrence of the *task-dependences* event to announce its dependences with respect to the list items in the **depend** clause. This callback has type signature **ompt_callback_dependences_t**.

A thread dispatches the **ompt_callback_task_dependence** callback for a *task-dependence* event to report a dependence between a predecessor task (*src_task_data*) and a dependent task (*sink_task_data*). This callback has type signature **ompt_callback_task_dependence_t**.

Restrictions

Restrictions to the **depend** clause are as follows:

- List items, other than reserved locators, used in **depend** clauses of the same task or sibling tasks must indicate identical storage locations or disjoint storage locations.
- List items used in **depend** clauses cannot be zero-length array sections.
- The **omp_all_memory** reserved locator can only be used in a **depend** clause with an **out** or **inout** *task-dependence-type*.
- Array sections cannot be specified in **depend** clauses with the **depobj** *task-dependence-type*.
- List items used in **depend** clauses with the **depobj** *task-dependence-type* must be depend objects in the initialized state.
- List items used in **depend** clauses with the **depobj** *task-dependence-type* must be expressions of the OpenMP **depend** type.
- List items that are expressions of the OpenMP **depend** type can only be used in **depend** clauses with the **depobj** *task-dependence-type*.

Fortran

- A common block name cannot appear in a **depend** clause.

Fortran

C / C++

- A bit-field cannot appear in a **depend** clause.

C / C++

Cross References

- **ompt_callback_dependences_t**, see Section 19.5.2.8.
- **ompt_callback_task_dependence_t**, see Section 19.5.2.9.
- Array sections, see Section 3.2.4.
- Array shaping, see Section 3.2.3.
- Task scheduling constraints, see Section 12.9.
- Iterators, see Section 3.2.5.
- **target** construct, see Section 13.8.
- **target enter data** construct, see Section 13.6.
- **target exit data** construct, see Section 13.7.
- **target update** construct, see Section 13.9.
- **task** construct, see Section 12.5.
- *task-dependence-modifiers* construct, see Section 15.9.1.
- **depobj** construct, see Section 15.9.4.

15.9.6 doacross Clause

Name:	**Properties:**
doacross	default

Arguments:

Name	Type	Properties
vector	loop-iteration vector	default

Modifiers:

Name	Modifies	Type	Properties
dependence-type	*vector*	**Keyword: sink, source**	ultimate, unique, required

Directives:

ordered

Additional information: The *clause-name* **depend** may be used as a synonym for the *clause-name* **doacross**. This use has been deprecated.

Semantics

The **doacross** clause identifies cross-iteration dependences that imply additional constraints on the scheduling of loop iterations. These constraints establish dependences only between loop iterations.

The **source** *dependence-type* specifies the satisfaction of cross-iteration dependences that arise from the current iteration. If the **source** *dependence-type* is specified then the *vector* argument is optional; if *vector* is omitted, it is assumed to be **omp_cur_iteration**.

The **sink** *dependence-type* specifies a cross-iteration dependence, where *vector* indicates the iteration that satisfies the dependence.

If *vector* does not occur in the iteration space, the **doacross** clause is ignored. If all **doacross** clauses on an **ordered** construct are ignored then the construct is ignored.

Note – If the **sink** *dependence-type* is specified for a *vector* that does not indicate an earlier iteration of the logical iteration space, deadlock may occur.

Restrictions

Restrictions to the **doacross** clause are as follows:

- At most one **doacross** clause can be specified on a directive with **source** as the *dependence-type*.
- The most closely nested loop-associated directive must specify an **ordered** clause and *vector* must have *n* dimensions, where *n* is the argument specified for that **ordered** clause or *vector* must be **omp_cur_iteration** if the **source** *dependence-type* is specified or *vector* must be **omp_cur_iteration** - 1 if the **sink** *dependence-type* is specified.
- If *vector* is specified with **source** as the *dependence-type* then it must be **omp_cur_iteration**.
- For each element of *vector* for which the **sink** *dependence-type* is specified, if the loop iteration variable var_i has an integral or pointer type, the i^{th} expression of *vector* must be computable without overflow in that type for any value of var_i that can encounter the construct on which the **doacross** clause appears.

C++

- For each element of *vector* for which the **sink** *dependence-type* is specified, if the loop iteration variable var_i is of a random access iterator type other than pointer type, the i^{th} expression of *vector* must be computable without overflow in the type that would be used by **std::distance** applied to variables of the type of var_i for any value of var_i that can encounter the construct on which the **doacross** clause appears.

C++

Cross References

- Loop-iteration vectors, see Section 4.4.2.
- **ordered** construct, see Section 15.9.7.

15.9.7 ordered Construct

Name: **ordered**	**Association:** none
Category: executable	**Properties:** simdizable, thread-limiting

Clause groups:
parallelization-level

Clauses:
doacross

Binding

The binding thread set for an **ordered** region is the current team. An **ordered** region binds to the innermost enclosing loop-associated region.

Semantics

The **ordered** construct specifies that execution must not violate cross-iteration dependences as specified in the clauses that appear on the construct. While the **ordered** construct is specified as a stand-alone directive, it may also be treated as a block-associated construct. If the construct is block-associated then the effect is as if an **ordered** construct with the same *parallelization-level* was specified at the location of the directive with a **doacross** clause with a **sink** *dependence-type* with a *vector* argument equal to **omp_cur_iteration** - 1 and an **ordered** construct with the same *parallelization-level* was specified at the end of the structured block (e.g., the location of the end directive when specified) with a **doacross** clause with a **source** *dependence-type* with no argument. If no clauses are specified, the construct must be block-associated and the effect is as if the **threads** *parallelization-level* clause was specified.

If the **threads** clause is specified, the threads in the team that is executing the worksharing-loop region execute **ordered** regions sequentially in the order of the loop iterations. If any **doacross** clauses are specified then those clauses specify the order in which the threads in the team execute **ordered** regions. If the **simd** clause is specified, the **ordered** regions encountered by any thread will execute one at a time in the order of the loop iterations. With either *parallelization-level*, execution of code outside the region for different iterations can run in parallel; execution of that code within the same iteration must observe any constraints imposed by the base-languge semantics.

When the thread that is executing the first iteration of the loop encounters an **ordered** construct, it can enter the **ordered** region without waiting. When a thread that is executing any subsequent iteration encounters an **ordered** construct without a **doacross** clause, it waits at the beginning of the **ordered** region until execution of all **ordered** regions that belong to all previous iterations has completed. When a thread that is executing any subsequent iteration encounters an

ordered construct with one or more **doacross** clauses for which the **sink** *dependence-type* is specified, the thread waits until its dependences on all valid iterations specified by the **doacross** clauses are satisfied before it continues execution. A specific dependence is satisfied when a thread that is executing the corresponding iteration encounters an **ordered** construct with a **doacross** clause for which the **source** *dependence-type* is specified.

ordered regions that bind to different regions execute independently of each other.

Execution Model Events

The *ordered-acquiring* event occurs in the task that encounters the **ordered** construct on entry to the ordered region before it initiates synchronization for the region.

The *ordered-acquired* event occurs in the task that encounters the **ordered** construct after it enters the region, but before it executes the structured block of the **ordered** region.

The *ordered-released* event occurs in the task that encounters the **ordered** construct after it completes any synchronization on exit from the **ordered** region.

The *doacross-sink* event occurs in the task that encounters an **ordered** construct for each **doacross** clause for which the **sink** *dependence-type* is specified after the dependence is fulfilled.

The *doacross-source* event occurs in the task that encounters an **ordered** construct with a **doacross** clause for which the **source** *dependence-type* is specified before signaling that the dependence has been fulfilled.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of an *ordered-acquiring* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of an *ordered-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_mutex_released** callback with **ompt_mutex_ordered** as the *kind* argument if practical, although a less specific kind may be used, for each occurrence of an *ordered-released* event in that thread. This callback has the type signature **ompt_callback_mutex_t** and occurs in the task that encounters the ordered construct.

A thread dispatches a registered **ompt_callback_dependences** callback with all vector entries listed as **ompt_dependence_type_sink** in the *deps* argument for each occurrence of a *doacross-sink* event in that thread. A thread dispatches a registered **ompt_callback_dependences** callback with all vector entries listed as **ompt_dependence_type_source** in the *deps* argument for each occurrence of a *doacross-source* event in that thread. These callbacks have the type signature **ompt_callback_dependences_t**.

Restrictions

Restrictions to the **ordered** construct are as follows:

- The construct is simdizable only if the **simd** *parallelization-level* is specified.
- If the **simd** *parallelization-level* is specified, the binding region must be a **simd** region or one that corresponds to a combined or composite construct for which the **simd** construct is a leaf construct.
- If the **threads** *parallelization-level* is specified, the binding region must be a *worksharing-loop* region or one that corresponds to a combined or composite construct for which the *worksharing-loop* is a leaf construct.
- If the **threads** *parallelization-level* is specified and the binding region corresponds to a combined or composite construct then **simd** construct must not be a leaf construct unless the **simd** *parallelization-level* is also specified.
- The construct that corresponds to the binding region of an **ordered** region specify an **ordered** clause.
- The construct that corresponds to the binding region of an **ordered** region must not specify a **reduction** clause with the **inscan** modifier.
- Either a **doacross** clause with a **sink** *dependence-type* or a **doacross** clause with a **source** *dependence-type* may appear on an **ordered** construct, but not both.
- A thread must not encounter more than one **ordered** region that corresponds to a block-associated **ordered** construct during execution of a logical iteration of the loop-associated construct to which the **ordered** construct binds.

Cross References

- **ompt_callback_mutex_acquire_t**, see Section 19.5.2.14.
- **ompt_callback_mutex_t**, see Section 19.5.2.15.
- **ompt_mutex_ordered**, see Section 19.4.4.17.
- Worksharing-loop construct, see Section 11.5.
- **doacross** clause, see Section 15.9.6
- *parallelization-type* clauses, see Section 15.9.8
- **simd** construct, see Section 10.4.

15.9.8 *parallelization-type* Clauses

Clause group: parallelization-level

Properties: unique, inarguable	**Members:** **simd**, **threads**

Semantics

The *parallelization-level* clause grouping defines a set of clauses that indicate the type of parallelization (**threads** or **simd**) with which to associate a construct.

Cross References

- **ordered** construct, see Section 15.9.7.

16 Cancellation Constructs

This chapter defines constructs related to cancellation of OpenMP regions.

16.1 `cancel` Construct

Name: **`cancel`** **Category:** executable	**Association:** none **Properties:** default

Clauses:
`cancel-directive-name`, **`if`**

Binding

The binding thread set of the **`cancel`** region is the current team. The binding region of the **`cancel`** region is the innermost enclosing region of the type that corresponds to *cancel-directive-name*.

Semantics

The **`cancel`** construct activates cancellation of the innermost enclosing region of the type specified. Cancellation of the binding region is activated only if the *cancel-var* ICV is *true*, in which case the **`cancel`** construct causes the encountering task to continue execution at the end of the binding region if *cancel-directive-name* is not **`taskgroup`**. If the *cancel-var* ICV is *true* and *cancel-directive-name* is **`taskgroup`**, the encountering task continues execution at the end of the current task region. If the *cancel-var* ICV is *false*, the **`cancel`** construct is ignored.

Threads check for active cancellation only at cancellation points that are implied at the following locations:

- **`cancel`** regions;
- **`cancellation point`** regions;
- **`barrier`** regions;
- at the end of a worksharing-loop construct with a **`nowait`** clause and for which the same list item appears in both **`firstprivate`** and **`lastprivate`** clauses; and
- implicit barrier regions.

When a thread reaches one of the above cancellation points and if the *cancel-var* ICV is *true*, then:

- If the thread is at a **cancel** or **cancellation point** region and *cancel-directive-name* is not **taskgroup**, the thread continues execution at the end of the canceled region if cancellation has been activated for the innermost enclosing region of the type specified.
- If the thread is at a **cancel** or **cancellation point** region and *cancel-directive-name* is **taskgroup**, the encountering task checks for active cancellation of all of the *taskgroup sets* to which the encountering task belongs, and continues execution at the end of the current task region if cancellation has been activated for any of the *taskgroup sets*.
- If the encountering task is at a barrier region or at the end of a worksharing-loop construct with a **nowait** clause and for which the same list item appears in both **firstprivate** and **lastprivate** clauses, the encountering task checks for active cancellation of the innermost enclosing **parallel** region. If cancellation has been activated, then the encountering task continues execution at the end of the canceled region.

Note – If one thread activates cancellation and another thread encounters a cancellation point, the order of execution between the two threads is non-deterministic. Whether the thread that encounters a cancellation point detects the activated cancellation depends on the underlying hardware and operating system.

When cancellation of tasks is activated through a **cancel** construct with **taskgroup** for *cancel-directive-name*, the tasks that belong to the *taskgroup set* of the innermost enclosing **taskgroup** region will be canceled. The task that encountered that construct continues execution at the end of its task region, which implies completion of that task. Any task that belongs to the innermost enclosing **taskgroup** and has already begun execution must run to completion or until a cancellation point is reached. Upon reaching a cancellation point and if cancellation is active, the task continues execution at the end of its task region, which implies the completion of the task. Any task that belongs to the innermost enclosing **taskgroup** and that has not begun execution may be discarded, which implies its completion.

When cancellation of tasks is activated through a **cancel** construct with *cancel-directive-name* other than **taskgroup**, each thread of the binding thread set resumes execution at the end of the canceled region if a cancellation point is encountered. If the canceled region is a parallel region, any tasks that have been created by a **task** or a **taskloop** construct and their descendant tasks are canceled according to the above **taskgroup** cancellation semantics. If the canceled region is not a parallel region, no task cancellation occurs.

C++

The usual C++ rules for object destruction are followed when cancellation is performed.

C++

Fortran

All private objects or subobjects with **ALLOCATABLE** attribute that are allocated inside the canceled construct are deallocated.

Fortran

If the canceled construct contains a **reduction**, **task_reduction** or **lastprivate** clause, the final values of the list items that appeared in those clauses are undefined.

When an **if** clause is present on a **cancel** construct and the **if** expression evaluates to *false*, the **cancel** construct does not activate cancellation. The cancellation point associated with the **cancel** construct is always encountered regardless of the value of the **if** expression.

Note – The programmer is responsible for releasing locks and other synchronization data structures that might cause a deadlock when a **cancel** construct is encountered and blocked threads cannot be canceled. The programmer is also responsible for ensuring proper synchronizations to avoid deadlocks that might arise from cancellation of OpenMP regions that contain OpenMP synchronization constructs.

Execution Model Events

If a task encounters a **cancel** construct that will activate cancellation then a *cancel* event occurs.

A *discarded-task* event occurs for any discarded tasks.

Tool Callbacks

A thread dispatches a registered **ompt_callback_cancel** callback for each occurrence of a *cancel* event in the context of the encountering task. This callback has type signature **ompt_callback_cancel_t**; **(***flags* **& ompt_cancel_activated)** always evaluates to *true* in the dispatched callback; **(***flags* **& ompt_cancel_parallel)** evaluates to *true* in the dispatched callback if *cancel-directive-name* is **parallel**; **(***flags* **& ompt_cancel_sections)** evaluates to *true* in the dispatched callback if *cancel-directive-name* is **sections**; **(***flags* **& ompt_cancel_loop)** evaluates to *true* in the dispatched callback if *cancel-directive-name* is **for** or **do**; and **(***flags* **& ompt_cancel_taskgroup)** evaluates to *true* in the dispatched callback if *cancel-directive-name* is **taskgroup**.

A thread dispatches a registered **ompt_callback_cancel** callback with the *ompt_data_t* associated with the discarded task as its *task_data* argument and **ompt_cancel_discarded_task** as its *flags* argument for each occurrence of a *discarded-task* event. The callback occurs in the context of the task that discards the task and has type signature **ompt_callback_cancel_t**.

Restrictions

Restrictions to the **cancel** construct are as follows:

- The behavior for concurrent cancellation of a region and a region nested within it is unspecified.
- If *cancel-directive-name* is **taskgroup**, the **cancel** construct must be closely nested inside a **task** or a **taskloop** construct and the **cancel** region must be closely nested inside a **taskgroup** region.
- If *cancel-directive-name* is **sections**, the **cancel** construct must be closely nested inside a **sections** or **section** construct.
- If *cancel-directive-name* is neither **sections** nor **taskgroup**, the **cancel** construct must be closely nested inside an OpenMP construct that matches *cancel-directive-name*.
- A worksharing construct that is canceled must not have a **nowait** clause or a **reduction** clause with a user-defined reduction that uses **omp_orig** in the *initializer-expr* of the corresponding **declare reduction** directive.
- A worksharing-loop construct that is canceled must not have an **ordered** clause or a **reduction** clause with the **inscan** modifier.
- When cancellation is active for a **parallel** region, a thread in the team that binds to that region may not be executing or encounter a worksharing construct with an **ordered** clause, a **reduction** clause with the **inscan** modifier or a **reduction** clause with a user-defined reduction that uses **omp_orig** in the *initializer-expr* of the corresponding **declare reduction** directive.
- When cancellation is active for a **parallel** region, a thread in the team that binds to that region may not be executing or encounter a **scope** construct with a **reduction** clause with a user-defined reduction that uses **omp_orig** in the *initializer-expr* of the corresponding **declare reduction** directive.
- During execution of a construct that may be subject to cancellation, a thread must not encounter an orphaned cancellation point. That is, a cancellation point must only be encountered within that construct and must not be encountered elsewhere in its region.

Cross References

- **if** clause, see Section 3.4.
- **ompt_callback_cancel_t**, see Section 19.5.2.18.
- **omp_cancel_flag_t** enumeration type, see Section 19.4.4.26.
- *cancel-var* ICV, see Section 2.1.
- **cancellation point** construct, see Section 16.2.
- **omp_get_cancellation** routine, see Section 18.2.8.

16.2 `cancellation point` Construct

Name: `cancellation point`	**Association:** none
Category: executable	**Properties:** default

Clauses:
`cancel-directive-name`

Binding

The binding thread set of the **`cancellation point`** construct is the current team. The binding region of the **`cancellation point`** region is the innermost enclosing region of the type that corresponds to *cancel-directive-name*.

Semantics

The **`cancellation point`** construct introduces a user-defined cancellation point at which an implicit or explicit task must check if cancellation of the innermost enclosing region of the type specified has been activated. This construct does not implement any synchronization between threads or tasks. When an implicit or explicit task reaches a user-defined cancellation point and if the *cancel-var* ICV is *true*, then:

- If the *cancel-directive-name* of the encountered **`cancellation point`** construct is not **`taskgroup`**, the thread continues execution at the end of the canceled region if cancellation has been activated for the innermost enclosing region of the type specified.
- If the *cancel-directive-name* of the encountered **`cancellation point`** construct is **`taskgroup`**, the encountering task checks for active cancellation of all *taskgroup sets* to which the encountering task belongs and continues execution at the end of the current task region if cancellation has been activated for any of them.

Execution Model Events

The *cancellation* event occurs if a task encounters a cancellation point and detected the activation of cancellation.

Tool Callbacks

A thread dispatches a registered **`ompt_callback_cancel`** callback for each occurrence of a *cancel* event in the context of the encountering task. This callback has type signature **`ompt_callback_cancel_t`**; **(***flags* **& `ompt_cancel_detected`)** always evaluates to *true* in the dispatched callback; **(***flags* **& `ompt_cancel_parallel`)** evaluates to *true* in the dispatched callback if *cancel-directive-name* of the encountered **`cancellation point`** construct is **`parallel`**; **(***flags* **& `ompt_cancel_sections`)** evaluates to *true* in the dispatched callback if *cancel-directive-name* of the encountered **`cancellation point`** construct is **`sections`**; **(***flags* **& `ompt_cancel_loop`)** evaluates to *true* in the dispatched callback if *cancel-directive-name* of the encountered **`cancellation point`** construct is **`for`** or **`do`**; and **(***flags* **& `ompt_cancel_taskgroup`)** evaluates to *true* in the dispatched callback if *cancel-directive-name* of the encountered **`cancellation point`** construct is **`taskgroup`**.

Restrictions

Restrictions to the **cancellation point** construct are as follows:

- A **cancellation point** construct for which *cancel-directive-name* is **taskgroup** must be closely nested inside a **task** or **taskloop** construct, and the **cancellation point** region must be closely nested inside a **taskgroup** region.
- A **cancellation point** construct for which *cancel-directive-name* is **sections** must be closely nested inside a **sections** or **section** construct.
- A **cancellation point** construct for which *cancel-directive-name* is neither **sections** nor **taskgroup** must be closely nested inside an OpenMP construct that matches *cancel-directive-name*.

Cross References

- **ompt_callback_cancel_t**, see Section 19.5.2.18.
- *cancel-var* ICV, see Section 2.1.
- **cancel** construct, see Section 16.1.
- **omp_get_cancellation** routine, see Section 18.2.8.

17 Composition of Constructs

This chapter defines rules and mechanisms for nesting regions and for combining constructs.

17.1 Nesting of Regions

This section describes a set of restrictions on the nesting of regions. The restrictions on nesting are as follows:

- A worksharing region may not be closely nested inside a worksharing, **task**, **taskloop**, **critical**, **ordered**, **atomic**, or **masked** region.
- A **barrier** region may not be closely nested inside a worksharing, **task**, **taskloop**, **critical**, **ordered**, **atomic**, or **masked** region.
- A **masked** region may not be closely nested inside a worksharing, **atomic**, **task**, or **taskloop** region.
- An **ordered** region that corresponds to an **ordered** construct without any clause or with the **threads** or **depend** clause may not be closely nested inside a **critical**, **ordered**, **loop**, **atomic**, **task**, or **taskloop** region.
- An **ordered** region that corresponds to an **ordered** construct without the **simd** clause specified must be closely nested inside a worksharing-loop region.
- An **ordered** region that corresponds to an **ordered** construct with the **simd** clause specified must be closely nested inside a **simd** or worksharing-loop SIMD region.
- An **ordered** region that corresponds to an **ordered** construct with both the **simd** and **threads** clauses must be closely nested inside a worksharing-loop SIMD region or closely nested inside a worksharing-loop and **simd** region.
- A **critical** region may not be nested (closely or otherwise) inside a **critical** region with the same name. This restriction is not sufficient to prevent deadlock.
- OpenMP constructs may not be encountered during execution of an **atomic** region.
- The only OpenMP constructs that can be encountered during execution of a **simd** (or worksharing-loop SIMD) region are the **atomic** construct, the **loop** construct without a defined binding region, the **simd** construct and the **ordered** construct with the **simd** clause.

- If a **target update**, **target data**, **target enter data**, or **target exit data** construct is encountered during execution of a **target** region, the behavior is unspecified.
- If a **target** construct is encountered during execution of a **target** region and a **device** clause in which the **ancestor** *device-modifier* appears is not present on the construct, the behavior is unspecified.
- A **teams** region must be strictly nested either within the implicit parallel region that surrounds the whole OpenMP program or within a **target** region. If a **teams** construct is nested within a **target** construct, that **target** construct must contain no statements, declarations or directives outside of the **teams** construct.
- **distribute** regions, including any **distribute** regions arising from composite constructs, **parallel** regions, including any **parallel** regions arising from combined constructs, **loop** regions, **omp_get_num_teams()** regions, and **omp_get_team_num()** regions are the only OpenMP regions that may be strictly nested inside the **teams** region.
- A **loop** region that binds to a **teams** region must be strictly nested inside a **teams** region.
- A **distribute** region must be strictly nested inside a **teams** region.
- If *construct-type-clause* is **taskgroup**, the **cancel** construct must be closely nested inside a **task** construct and the **cancel** region must be closely nested inside a **taskgroup** region. If *construct-type-clause* is **sections**, the **cancel** construct must be closely nested inside a **sections** or **section** construct. Otherwise, the **cancel** construct must be closely nested inside an OpenMP construct that matches the type specified in *construct-type-clause* of the **cancel** construct.
- A **cancellation point** construct for which *construct-type-clause* is **taskgroup** must be closely nested inside a **task** construct, and the **cancellation point** region must be closely nested inside a **taskgroup** region. A **cancellation point** construct for which *construct-type-clause* is **sections** must be closely nested inside a **sections** or **section** construct. Otherwise, a **cancellation point** construct must be closely nested inside an OpenMP construct that matches the type specified in *construct-type-clause*.
- The only constructs that may be encountered inside a region that corresponds to a construct with an **order** clause that specifies **concurrent** are the **loop** construct, the **parallel** construct, the **simd** construct, and combined constructs for which the first construct is a **parallel** construct.
- A region that corresponds to a construct with an **order** clause that specifies **concurrent** may not contain calls to procedures that contain OpenMP directives or calls to the OpenMP Runtime API.

17.2 Clauses on Combined and Composite Constructs

This section specifies the handling of clauses on combined or composite constructs and the handling of implicit clauses from variables with predetermined data sharing if they are not predetermined only on a particular construct. Some clauses are permitted only on a single leaf construct of the combined or composite construct, in which case the effect is as if the clause is applied to that specific construct. Other clauses that are permitted on more than one leaf construct have the effect as if they are applied to a subset of those constructs, as detailed in this section.

The **collapse** clause is applied once to the combined or composite construct.

The effect of the **private** clause is as if it is applied only to the innermost leaf construct that permits it.

The effect of the **firstprivate** clause is as if it is applied to one or more leaf constructs as follows:

- To the **distribute** construct if it is among the constituent constructs;
- To the **teams** construct if it is among the constituent constructs and the **distribute** construct is not;
- To a worksharing construct that accepts the clause if one is among the constituent constructs;
- To the **taskloop** construct if it is among the constituent constructs;
- To the **parallel** construct if it is among the constituent constructs and neither a **taskloop** construct nor a worksharing construct that accepts the clause is among them;
- To the **target** construct if it is among the constituent constructs and the same list item neither appears in a **lastprivate** clause nor is the base variable or base pointer of a list item that appears in a **map** clause.

If the **parallel** construct is among the constituent constructs and the effect is not as if the **firstprivate** clause is applied to it by the above rules, then the effect is as if the **shared** clause with the same list item is applied to the **parallel** construct. If the **teams** construct is among the constituent constructs and the effect is not as if the **firstprivate** clause is applied to it by the above rules, then the effect is as if the **shared** clause with the same list item is applied to the **teams** construct.

The effect of the **lastprivate** clause is as if it is applied to all leaf constructs that permit the clause. If the **parallel** construct is among the constituent constructs and the list item is not also specified in the **firstprivate** clause, then the effect of the **lastprivate** clause is as if the **shared** clause with the same list item is applied to the **parallel** construct. If the **teams** construct is among the constituent constructs and the list item is not also specified in the **firstprivate** clause, then the effect of the **lastprivate** clause is as if the **shared** clause with the same list item is applied to the **teams** construct. If the **target** construct is among the constituent constructs and the list item is not the base variable or base pointer of a list item that appears in a **map** clause, the effect of the **lastprivate** clause is as if the same list item appears in a **map** clause with a *map-type* of **tofrom**.

The effect of the **shared**, **default**, **order**, or **allocate** clause is as if it is applied to all leaf constructs that permit the clause.

The effect of the **reduction** clause is as if it is applied to all leaf constructs that permit the clause, except for the following constructs:

- The **parallel** construct, when combined with the **sections**, worksharing-loop, **loop**, or **taskloop** construct; and
- The **teams** construct, when combined with the **loop** construct.

For the **parallel** and **teams** constructs above, the effect of the **reduction** clause instead is as if each list item or, for any list item that is an array item, its corresponding base array or base pointer appears in a **shared** clause for the construct. If the **task** *reduction-modifier* is specified, the effect is as if it only modifies the behavior of the **reduction** clause on the innermost leaf construct that accepts the modifier (see Section 5.5.9). If the **inscan** *reduction-modifier* is specified, the effect is as if it modifies the behavior of the **reduction** clause on all constructs of the combined construct to which the clause is applied and that accept the modifier. If a list item in a **reduction** clause on a combined target construct does not have the same base variable or base pointer as a list item in a **map** clause on the construct, then the effect is as if the list item in the **reduction** clause appears as a list item in a **map** clause with a *map-type* of **tofrom**.

The effect of the **if** clause is described in Section 3.4.

The effect of the **linear** clause is as if it is applied to the innermost leaf construct. Additionally, if the list item is not the iteration variable of a **simd** or worksharing-loop SIMD construct, the effect on the outer leaf constructs is as if the list item was specified in **firstprivate** and **lastprivate** clauses on the combined or composite construct, with the rules specified above applied. If a list item of the **linear** clause is the iteration variable of a **simd** or worksharing-loop SIMD construct and it is not declared in the construct, the effect on the outer leaf constructs is as if the list item was specified in a **lastprivate** clause on the combined or composite construct with the rules specified above applied.

The effect of the **nowait** clause is as if it is applied to the outermost leaf construct that permits it.

If the clauses have expressions on them, such as for various clauses where the argument of the clause is an expression, or *lower-bound*, *length*, or *stride* expressions inside array sections (or *subscript* and *stride* expressions in *subscript-triplet* for Fortran), or *linear-step* or *alignment* expressions, the expressions are evaluated immediately before the construct to which the clause has been split or duplicated per the above rules (therefore inside of the outer leaf constructs). However, the expressions inside the **num_teams** and **thread_limit** clauses are always evaluated before the outermost leaf construct.

The restriction that a list item may not appear in more than one data sharing clause with the exception of specifying a variable in both **firstprivate** and **lastprivate** clauses applies after the clauses are split or duplicated per the above rules.

Restrictions

Restrictions to clauses on combined and composite constructs are as follows:

- A clause that appears on a combined or composite construct must apply to at least one of the leaf constructs per the rules defined in this section.

17.3 Combined and Composite Directive Names

Combined constructs are shortcuts for specifying one construct immediately nested inside another construct. Composite constructs are also shortcuts for specifying the effect of one construct immediately following the effect of another construct. However, composite constructs define semantics to combine constructs that cannot otherwise be immediately nested.

For all combined and composite constructs, *directive-name* concatenates *directive-name-A*, the directive name of the enclosing construct, with an intervening space followed by *directive-name-B*, the directive name of the nested construct. If *directive-name-A* and *directive-name-B* both correspond to loop-associated constructs then *directive-name* is a composite construct. Otherwise *directive-name* is a combined construct.

If *directive-name-A* is **taskloop** or the directive name of a worksharing-loop construct then *directive-name-B* may be **simd**.

If *directive-name-A* is **masked** then *directive-name-B* may be **taskloop** or the directive name of a combined or composite construct for which *directive-name-A* is **taskloop**.

If *directive-name-A* is **parallel** then *directive-name-B* may be **loop**, **sections**, **workshare**, **masked**, the directive name of a worksharing-loop construct or the directive name of a combined or composite construct for which *directive-name-A* is **masked** or the directive name of a worksharing-loop construct.

If *directive-name-A* is **distribute** then *directive-name-B* may be **simd** or the directive name of a combined or composite construct for which *directive-name-A* is **parallel** and a worksharing-loop construct is a leaf construct.

If *directive-name-A* is **teams** then *directive-name-B* may be **loop**, **distribute** or the directive name of a combined or composite construct for which *directive-name-A* is **distribute**.

If *directive-name-A* is **target** then *directive-name-B* may be **simd**, **parallel**, **teams**, the directive name of a combined or composite construct for which *directive-name-A* is **teams** or the directive name of a combined or composite construct for which *directive-name-A* is **parallel** and **loop** or a worksharing-loop construct is a leaf construct.

For all combined or composite constructs for which the **masked** construct is a leaf construct, the directive name **master** may be substituted for the directive name **masked**. The use of the directive name **master** has been deprecated.

Cross References

- **masked** construct, see Section 10.5.
- **parallel** construct, see Section 10.1.
- **teams** construct, see Section 10.2.
- Worksharing-loop construct, see Section 11.5.
- **distribute** construct, see Section 11.6.
- **loop** construct, see Section 11.7.
- **sections** construct, see Section 11.3.
- **target** construct, see Section 13.8.
- **taskloop** construct, see Section 12.6.
- **workshare** construct, see Section 11.4.

17.4 Combined Construct Semantics

The semantics of the combined constructs are identical to that of explicitly specifying the first construct containing one instance of the second construct and no other statements. All combined and composite directives for which a loop-associated construct is a leaf construct are themselves loop-associated constructs. For combined constructs, tool callbacks are invoked as if the constructs were explicitly nested.

Restrictions

Restrictions to combined constructs are as follows:

- The restrictions of *directive-name-A* and *directive-name-B* apply.
- If *directive-name-A* is **parallel**, the **nowait** and **in_reduction** clauses must not be specified.
- If *directive-name-A* is **target**, the **copyin** clause must not be specified.

Cross References

- **nowait** clause, see Section 15.6.
- **parallel** construct, see Section 10.1.
- **copyin** clause, see Section 5.7.1.
- **in_reduction** clause, see Section 5.5.11.

17.5 Composite Construct Semantics

Composite constructs combine constructs that otherwise cannot be immediately nested. Specifically, composite constructs apply multiple loop-associated constructs to the same canonical loop nest. The semantics of each composite construct first apply the semantics of the enclosing construct as specified by *directive-name-A* and any clauses that apply to it. For each task (possibly implicit, possibly initial) as appropriate for the semantics of *directive-name-A*, the application of its semantics yields a nested loop of depth two in which the outer loop iterates over the chunks assigned to that task and the inner loop iterates over the logical iterations of each chunk. The semantics of *directive-name-B* and any clauses that apply to it are then applied to that inner loop. For composite constructs, tool callbacks are invoked as if the constructs were explicitly nested.

If *directive-name-A* is **taskloop** and *directive-name-B* is **simd** then for the application of the **simd** construct, the effect of any **in_reduction** clause is as if a **reduction** clause with the same reduction operator and list items is present.

Restrictions

Restrictions to composite constructs are as follows:

- The restrictions of *directive-name-A* and *directive-name-B* apply.
- If *directive-name-A* is **distribute**, the **linear** clause may only be specified for loop iteration variables of loops that are associated with the construct.
- If *directive-name-A* is **distribute**, the **ordered** clause must not be specified.

Cross References

- Canonical loop nest form, see Section 4.4.1.
- Worksharing-loop construct, see Section 11.5.
- **distribute** construct, see Section 11.6.
- **firstprivate** clause, see Section 5.4.4.
- **in_reduction** clause, see Section 5.5.11.
- **lastprivate** clause, see Section 5.4.5.
- **linear** clause, see Section 5.4.6.
- **reduction** clause, see Section 5.5.9.
- **taskloop** construct, see Section 12.6.
- **simd** construct, see Section 10.4.

18 Runtime Library Routines

This chapter describes the OpenMP API runtime library routines and queryable runtime states. All OpenMP Runtime API names have an **omp_** prefix. Names that begin with the **ompx_** prefix are reserved for implementation-defined extensions to the OpenMP Runtime API. In this chapter, *true* and *false* are used as generic terms to simplify the description of the routines.

C / C++

true means a non-zero integer value and *false* means an integer value of zero.

C / C++

Fortran

true means a logical value of **.TRUE.** and *false* means a logical value of **.FALSE.**.

Fortran

Fortran

Restrictions

The following restrictions apply to all OpenMP runtime library routines:

- OpenMP runtime library routines may not be called from **PURE** or **ELEMENTAL** procedures.
- OpenMP runtime library routines may not be called in **DO CONCURRENT** constructs.

Fortran

18.1 Runtime Library Definitions

For each base language, a compliant implementation must supply a set of definitions for the OpenMP API runtime library routines and the special data types of their parameters. The set of definitions must contain a declaration for each OpenMP API runtime library routine and variable and a definition of each required data type listed below. In addition, each set of definitions may specify other implementation specific values.

C / C++

The library routines are external functions with "C" linkage.

Prototypes for the C/C++ runtime library routines described in this chapter shall be provided in a header file named **omp.h**. This file also defines the following:

- The type **omp_allocator_handle_t**, which must be an implementation-defined (for C++ possibly scoped) enum type with at least the **omp_null_allocator** enumerator with the value zero and an enumerator for each predefined memory allocator in Table 6.3;
- **omp_atv_default**, which is an instance of a type compatible with **omp_uintptr_t** with the value -1;
- The type **omp_control_tool_result_t**;
- The type **omp_control_tool_t**;
- The type **omp_depend_t**;
- The type **omp_event_handle_t**, which must be an implementation-defined (for C++ possibly scoped) enum type;
- The type **omp_intptr_t**, which is a signed integer type that is at least the size of a pointer on any device;
- The type **omp_interop_t**, which must be an implementation-defined integral or pointer type;
- The type **omp_interop_fr_t**, which must be an implementation-defined enum type with enumerators named **omp_ifr_*name*** where *name* is a foreign runtime name that is defined in the *OpenMP Additional Definitions* document;
- The type **omp_lock_hint_t** (deprecated);
- The type **omp_lock_t**;
- The type **omp_memspace_handle_t**, which must be an implementation-defined (for C++ possibly scoped) enum type with an enumerator for at least each predefined memory space in Table 6.1;
- The type **omp_nest_lock_t**;
- The type **omp_pause_resource_t**;
- The type **omp_proc_bind_t**;

- The type **omp_sched_t**;
- The type **omp_sync_hint_t**; and
- The type **omp_uintptr_t**, which is an unsigned integer type capable of holding a pointer on any device.
- The enumerator **omp_initial_device** with value negative one;
- The enumerator **omp_invalid_device** with an implementation-defined value less than negative one.

C / C++

C++

The OpenMP enumeration types provided in the **omp.h** header file shall not be scoped enumeration types unless explicitly allowed.

The **omp.h** header file also defines a class template that models the **Allocator** concept in the **omp::allocator** namespace for each predefined memory allocator in Table 6.3 for which the name includes neither the **omp_** prefix nor the **_alloc** suffix.

C++

Fortran

The OpenMP Fortran API runtime library routines are external procedures. The return values of these routines are of default kind, unless otherwise specified.

Interface declarations for the OpenMP Fortran runtime library routines described in this chapter shall be provided in the form of a Fortran **module** named **omp_lib** or a Fortran **include** file named **omp_lib.h**. Whether the **omp_lib.h** file provides derived-type definitions or those routines that require an explicit interface is implementation defined. Whether the **include** file or the **module** file (or both) is provided is also implementation defined.

These files also define the following:

- The default integer named constant **omp_allocator_handle_kind**;
- An integer named constant of kind **omp_allocator_handle_kind** for each predefined memory allocator in Table 6.3;
- The default integer named constant **omp_alloctrait_key_kind**;
- The default integer named constant **omp_alloctrait_val_kind**;
- The default integer named constant **omp_control_tool_kind**;
- The default integer named constant **omp_control_tool_result_kind**;
- The default integer named constant **omp_depend_kind**;
- The default integer named constant **omp_event_handle_kind**;
- The default integer named constant **omp_interop_kind**;

- The default integer named constant **omp_interop_fr_kind**;
- An integer named constant **omp_ifr_*name*** of kind **omp_interop_fr_kind** for each *name* that is a foreign runtime name that is defined in the *OpenMP Additional Definitions* document;
- The default integer named constant **omp_lock_hint_kind** (deprecated);
- The default integer named constant **omp_lock_kind**;
- The default integer named constant **omp_memspace_handle_kind**;
- An integer named constant of kind **omp_memspace_handle_kind** for each predefined memory space in Table 6.1;
- The default integer named constant **omp_nest_lock_kind**;
- The default integer named constant **omp_pause_resource_kind**;
- The default integer named constant **omp_proc_bind_kind**;
- The default integer named constant **omp_sched_kind**;
- The default integer named constant **omp_sync_hint_kind**;
- The default integer named constant **omp_initial_device** with value negative one;
- The default integer named constant **omp_invalid_device** with an implementation-defined value less than negative one; and
- The default integer named constant **openmp_version** with a value *yyyymm* where *yyyy* and *mm* are the year and month designations of the version of the OpenMP Fortran API that the implementation supports; this value matches that of the C preprocessor macro **_OPENMP**, when a macro preprocessor is supported (see Section 3.3).

Whether any of the OpenMP runtime library routines that take an argument are extended with a generic interface so arguments of different **KIND** type can be accommodated is implementation defined.

Fortran

18.2 Thread Team Routines

This section describes routines that affect and monitor thread teams in the current contention group.

18.2.1 omp_set_num_threads

Summary

The **omp_set_num_threads** routine affects the number of threads to be used for subsequent parallel regions that do not specify a **num_threads** clause, by setting the value of the first element of the *nthreads-var* ICV of the current task.

Format

C / C++

```
void omp_set_num_threads(int num_threads);
```

C / C++

Fortran

```
subroutine omp_set_num_threads(num_threads)
integer num_threads
```

Fortran

Constraints on Arguments

The value of the argument passed to this routine must evaluate to a positive integer, or else the behavior of this routine is implementation defined.

Binding

The binding task set for an **omp_set_num_threads** region is the generating task.

Effect

The effect of this routine is to set the value of the first element of the *nthreads-var* ICV of the current task to the value specified in the argument.

Cross References

- *nthreads-var* ICV, see Section 2.
- **OMP_NUM_THREADS** environment variable, see Section 21.1.2.
- **parallel** construct and **num_threads** clause, see Section 10.1.
- Determining the number of threads for a **parallel** region, see Section 10.1.1.
- **omp_get_max_threads** routine, see Section 18.2.3.
- **omp_get_num_threads** routine, see Section 18.2.2.

18.2.2 omp_get_num_threads

Summary

The **omp_get_num_threads** routine returns the number of threads in the current team.

Format

C / C++

```
int omp_get_num_threads(void);
```

C / C++

Fortran

```
integer function omp_get_num_threads()
```

Fortran

Binding

The binding region for an **omp_get_num_threads** region is the innermost enclosing **parallel** region.

Effect

The **omp_get_num_threads** routine returns the number of threads in the team that is executing the **parallel** region to which the routine region binds. If called from the sequential part of a program, this routine returns 1.

Cross References

- *nthreads-var* ICV, see Section 2.
- **OMP_NUM_THREADS** environment variable, see Section 21.1.2.
- **parallel** construct and **num_threads** clause, see Section 10.1.
- Determining the number of threads for a **parallel** region, see Section 10.1.1.
- **omp_set_num_threads** routine, see Section 18.2.1.

18.2.3 omp_get_max_threads

Summary

The **omp_get_max_threads** routine returns an upper bound on the number of threads that could be used to form a new team if a **parallel** construct without a **num_threads** clause were encountered after execution returns from this routine.

Format

C / C++

```c
int omp_get_max_threads(void);
```

C / C++

Fortran

```fortran
integer function omp_get_max_threads()
```

Fortran

Binding

The binding task set for an **omp_get_max_threads** region is the generating task.

Effect

The value returned by **omp_get_max_threads** is the value of the first element of the *nthreads-var* ICV of the current task. This value is also an upper bound on the number of threads that could be used to form a new team if a parallel region without a **num_threads** clause were encountered after execution returns from this routine.

Note – The return value of the **omp_get_max_threads** routine can be used to allocate sufficient storage dynamically for all threads in the team formed at the subsequent active **parallel** region.

Cross References

- *nthreads-var* ICV, see Section 2.
- **OMP_NUM_THREADS** environment variable, see Section 21.1.2.
- **parallel** construct and **num_threads** clause, see Section 10.1.
- Determining the number of threads for a **parallel** region, see Section 10.1.1.
- **omp_get_num_threads** routine, see Section 18.2.2.
- **omp_get_thread_num** routine, see Section 18.2.4.
- **omp_set_num_threads** routine, see Section 18.2.1.

18.2.4 omp_get_thread_num

Summary

The **omp_get_thread_num** routine returns the thread number, within the current team, of the calling thread.

Format

C / C++

```
int omp_get_thread_num(void);
```

C / C++

Fortran

```
integer function omp_get_thread_num()
```

Fortran

Binding

The binding thread set for an **omp_get_thread_num** region is the current team. The binding region for an **omp_get_thread_num** region is the innermost enclosing **parallel** region.

Effect

The **omp_get_thread_num** routine returns the thread number of the calling thread, within the team that is executing the **parallel** region to which the routine region binds. The thread number is an integer between 0 and one less than the value returned by **omp_get_num_threads**, inclusive. The thread number of the primary thread of the team is 0. The routine returns 0 if it is called from the sequential part of a program.

> Note – The thread number may change during the execution of an untied task. The value returned by **omp_get_thread_num** is not generally useful during the execution of such a task region.

Cross References

- *nthreads-var* ICV, see Section 2.
- **OMP_NUM_THREADS** environment variable, see Section 21.1.2.
- **parallel** construct and **num_threads** clause, see Section 10.1.
- Determining the number of threads for a **parallel** region, see Section 10.1.1.
- **omp_get_num_threads** routine, see Section 18.2.2.
- **omp_set_num_threads** routine, see Section 18.2.1.

18.2.5 omp_in_parallel

Summary

The **omp_in_parallel** routine returns *true* if the *active-levels-var* ICV is greater than zero; otherwise, it returns *false*.

Format

C / C++

```
int omp_in_parallel(void);
```

C / C++

Fortran

```
logical function omp_in_parallel()
```

Fortran

Binding

The binding task set for an **omp_in_parallel** region is the generating task.

Effect

The effect of the **omp_in_parallel** routine is to return *true* if the current task is enclosed by an active **parallel** region, and the **parallel** region is enclosed by the outermost initial task region on the device; otherwise it returns *false*.

Cross References

- *active-levels-var*, see Section 2.
- **parallel** construct, see Section 10.1.
- **omp_get_active_level** routine, see Section 18.2.20.
- **omp_get_num_threads** routine, see Section 18.2.2.

18.2.6 omp_set_dynamic

Summary

The **omp_set_dynamic** routine enables or disables dynamic adjustment of the number of threads available for the execution of subsequent **parallel** regions by setting the value of the *dyn-var* ICV.

Format

C / C++

```
void omp_set_dynamic(int dynamic_threads);
```

C / C++

Fortran

```
subroutine omp_set_dynamic(dynamic_threads)
logical dynamic_threads
```

Fortran

Binding

The binding task set for an **omp_set_dynamic** region is the generating task.

Effect

For implementations that support dynamic adjustment of the number of threads, if the argument to **omp_set_dynamic** evaluates to *true*, dynamic adjustment is enabled for the current task; otherwise, dynamic adjustment is disabled for the current task. For implementations that do not support dynamic adjustment of the number of threads, this routine has no effect: the value of *dyn-var* remains *false*.

Cross References

- *dyn-var* ICV, see Section 2.
- **OMP_DYNAMIC** environment variable, see Section 21.1.1.
- Determining the number of threads for a **parallel** region, see Section 10.1.1.
- **omp_get_dynamic** routine, see Section 18.2.7.
- **omp_get_num_threads** routine, see Section 18.2.2.

18.2.7 omp_get_dynamic

Summary

The **omp_get_dynamic** routine returns the value of the *dyn-var* ICV, which determines whether dynamic adjustment of the number of threads is enabled or disabled.

Format

C / C++

```
int omp_get_dynamic(void);
```

C / C++

Fortran

```
logical function omp_get_dynamic()
```

Fortran

Binding

The binding task set for an **omp_get_dynamic** region is the generating task.

Effect

This routine returns *true* if dynamic adjustment of the number of threads is enabled for the current task; it returns *false*, otherwise. If an implementation does not support dynamic adjustment of the number of threads, then this routine always returns *false*.

Cross References

- *dyn-var* ICV, see Section 2.
- **OMP_DYNAMIC** environment variable, see Section 21.1.1.
- Determining the number of threads for a **parallel** region, see Section 10.1.1.
- **omp_set_dynamic** routine, see Section 18.2.6.

18.2.8 omp_get_cancellation

Summary

The **omp_get_cancellation** routine returns the value of the *cancel-var* ICV, which determines if cancellation is enabled or disabled.

Format

C / C++

```
int omp_get_cancellation(void);
```

C / C++

Fortran

```
logical function omp_get_cancellation()
```

Fortran

Binding

The binding task set for an **omp_get_cancellation** region is the whole program.

Effect

This routine returns *true* if cancellation is enabled. It returns *false* otherwise.

Cross References

- **OMP_CANCELLATION** environment variable, see Section 21.2.6.
- *cancel-var* ICV, see Section 2.1.
- **cancel** construct, see Section 16.1.

18.2.9 omp_set_nested (Deprecated)

Summary

The deprecated **omp_set_nested** routine enables or disables nested parallelism by setting the *max-active-levels-var* ICV.

Format

C / C++

```
void omp_set_nested(int nested);
```

C / C++

Fortran

```
subroutine omp_set_nested(nested)
logical nested
```

Fortran

Binding

The binding task set for an **omp_set_nested** region is the generating task.

Effect

If the argument to **omp_set_nested** evaluates to *true*, the value of the *max-active-levels-var* ICV is set to the number of active levels of parallelism that the implementation supports; otherwise, if the value of *max-active-levels-var* is greater than 1 then it is set to 1. This routine has been deprecated.

Cross References

- *max-active-levels-var* ICV, see Section 2.
- **OMP_NESTED** environment variable, see Section 21.1.5.
- Determining the number of threads for a **parallel** region, see Section 10.1.1.
- **omp_get_max_active_levels** routine, see Section 18.2.16.
- **omp_get_nested** routine, see Section 18.2.10.
- **omp_set_max_active_levels** routine, see Section 18.2.15.

18.2.10 omp_get_nested (Deprecated)

Summary

The deprecated **omp_get_nested** routine returns whether nested parallelism is enabled or disabled, according to the value of the *max-active-levels-var* ICV.

Format

C / C++

```
int omp_get_nested(void);
```

C / C++

Fortran

```
logical function omp_get_nested()
```

Fortran

Binding

The binding task set for an **omp_get_nested** region is the generating task.

Effect

This routine returns *true* if *max-active-levels-var* is greater than 1 and greater than *active-levels-var* for the current task; it returns *false*, otherwise. If an implementation does not support nested parallelism, this routine always returns *false*. This routine has been deprecated.

Cross References

- *max-active-levels-var* ICV, see Section 2.
- **OMP_NESTED** environment variable, see Section 21.1.5.
- Determining the number of threads for a **parallel** region, see Section 10.1.1.
- **omp_get_max_active_levels** routine, see Section 18.2.16.
- **omp_set_max_active_levels** routine, see Section 18.2.15.
- **omp_set_nested** routine, see Section 18.2.9.

18.2.11 omp_set_schedule

Summary

The **omp_set_schedule** routine affects the schedule that is applied when **runtime** is used as schedule kind, by setting the value of the *run-sched-var* ICV.

Format

C / C++

```
void omp_set_schedule(omp_sched_t kind, int chunk_size);
```

C / C++

Fortran

```
subroutine omp_set_schedule(kind, chunk_size)
integer (kind=omp_sched_kind) kind
integer chunk_size
```

Fortran

Constraints on Arguments

The first argument passed to this routine can be one of the valid OpenMP schedule kinds (except for **runtime**) or any implementation-specific schedule. The C/C++ header file (**omp.h**) and the Fortran include file (**omp_lib.h**) and/or Fortran module file (**omp_lib**) define the valid constants. The valid constants must include the following, which can be extended with implementation-specific values:

C / C++

```c
typedef enum omp_sched_t {
  // schedule kinds
  omp_sched_static = 0x1,
  omp_sched_dynamic = 0x2,
  omp_sched_guided = 0x3,
  omp_sched_auto = 0x4,

  // schedule modifier
  omp_sched_monotonic = 0x80000000u
} omp_sched_t;
```

C / C++

Fortran

```fortran
! schedule kinds
integer(kind=omp_sched_kind), &
  parameter :: omp_sched_static = &
                   int(Z'1', kind=omp_sched_kind)
integer(kind=omp_sched_kind), &
  parameter :: omp_sched_dynamic = &
                   int(Z'2', kind=omp_sched_kind)
integer(kind=omp_sched_kind), &
  parameter :: omp_sched_guided = &
                   int(Z'3', kind=omp_sched_kind)
integer(kind=omp_sched_kind), &
  parameter :: omp_sched__auto = &
                   int(Z'4', kind=omp_sched_kind)

! schedule modifier
integer(kind=omp_sched_kind), &
  parameter :: omp_sched_monotonic = &
                   int(Z'80000000', kind=omp_sched_kind)
```

Fortran

Binding

The binding task set for an **omp_set_schedule** region is the generating task.

Effect

The effect of this routine is to set the value of the *run-sched-var* ICV of the current task to the values specified in the two arguments. The schedule is set to the schedule kind that is specified by the first argument *kind*. It can be any of the standard schedule kinds or any other implementation-specific one. For the schedule kinds **static**, **dynamic**, and **guided** the *chunk_size* is set to the value of the second argument, or to the default *chunk_size* if the value of the second argument is less than 1; for the schedule kind **auto** the second argument has no meaning; for implementation-specific schedule kinds, the values and associated meanings of the second argument are implementation defined.

Each of the schedule kinds can be combined with the **omp_sched_monotonic** modifier by using the + or | operators in C/C++ or the + operator in Fortran. If the schedule kind is combined with the **omp_sched_monotonic** modifier, the schedule is modified as if the **monotonic** schedule modifier was specified. Otherwise, the schedule modifier is **nonmonotonic**.

Cross References

- *run-sched-var* ICV, see Section 2.
- **OMP_SCHEDULE** environment variable, see Section 21.2.1.
- **omp_get_schedule** routine, see Section 18.2.12.
- **schedule** clause, see Section 11.5.3.

18.2.12 omp_get_schedule

Summary

The **omp_get_schedule** routine returns the schedule that is applied when the runtime schedule is used.

Format

C / C++

```
void omp_get_schedule(omp_sched_t *kind, int *chunk_size);
```

C / C++

Fortran

```
subroutine omp_get_schedule(kind, chunk_size)
integer (kind=omp_sched_kind) kind
integer chunk_size
```

Fortran

Binding

The binding task set for an **omp_get_schedule** region is the generating task.

Effect

This routine returns the *run-sched-var* ICV in the task to which the routine binds. The first argument *kind* returns the schedule to be used. It can be any of the standard schedule kinds as defined in Section 18.2.11, or any implementation-specific schedule kind. The second argument *chunk_size* returns the chunk size to be used, or a value less than 1 if the default chunk size is to be used, if the returned schedule kind is **static**, **dynamic**, or **guided**. The value returned by the second argument is implementation defined for any other schedule kinds.

Cross References

- *run-sched-var* ICV, see Section 2.
- **OMP_SCHEDULE** environment variable, see Section 21.2.1.
- **omp_set_schedule** routine, see Section 18.2.11.

18.2.13 omp_get_thread_limit

Summary

The **omp_get_thread_limit** routine returns the maximum number of OpenMP threads available to participate in the current contention group.

Format

C / C++

```
int omp_get_thread_limit(void);
```

C / C++

Fortran

```
integer function omp_get_thread_limit()
```

Fortran

Binding

The binding task set for an **omp_get_thread_limit** region is the generating task.

Effect

The **omp_get_thread_limit** routine returns the value of the *thread-limit-var* ICV.

Cross References

- *thread-limit-var* ICV, see Section 2.
- **OMP_NUM_THREADS** environment variable, see Section 21.1.2.
- **OMP_THREAD_LIMIT** environment variable, see Section 21.1.3.
- **omp_get_num_threads** routine, see Section 18.2.2.

18.2.14 omp_get_supported_active_levels

Summary

The **omp_get_supported_active_levels** routine returns the number of active levels of parallelism supported by the implementation.

Format

C / C++

```
int omp_get_supported_active_levels(void);
```

C / C++

Fortran

```
integer function omp_get_supported_active_levels()
```

Fortran

Binding

The binding task set for an **omp_get_supported_active_levels** region is the generating task.

Effect

The **omp_get_supported_active_levels** routine returns the number of active levels of parallelism supported by the implementation. The *max-active-levels-var* ICV may not have a value that is greater than this number. The value returned by the **omp_get_supported_active_levels** routine is implementation defined, but it must be greater than 0.

Cross References

- *max-active-levels-var* ICV, see Section 2.
- **omp_get_max_active_levels** routine, see Section 18.2.16.
- **omp_set_max_active_levels** routine, see Section 18.2.15.

18.2.15 omp_set_max_active_levels

Summary

The **omp_set_max_active_levels** routine limits the number of nested active parallel regions when a new nested parallel region is generated by the current task by setting the *max-active-levels-var* ICV.

Format

C / C++

```
void omp_set_max_active_levels(int max_levels);
```

C / C++

Fortran

```
subroutine omp_set_max_active_levels(max_levels)
integer max_levels
```

Fortran

Constraints on Arguments

The value of the argument passed to this routine must evaluate to a non-negative integer, otherwise the behavior of this routine is implementation defined.

Binding

The binding task set for an **omp_set_max_active_levels** region is the generating task.

Effect

The effect of this routine is to set the value of the *max-active-levels-var* ICV to the value specified in the argument.

If the number of active levels requested exceeds the number of active levels of parallelism supported by the implementation, the value of the *max-active-levels-var* ICV will be set to the number of active levels supported by the implementation.

Cross References

- *max-active-levels-var* ICV, see Section 2.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 21.1.4.
- **parallel** construct, see Section 10.1.
- **omp_get_max_active_levels** routine, see Section 18.2.16.
- **omp_get_supported_active_levels** routine, see Section 18.2.14.

18.2.16 omp_get_max_active_levels

Summary

The **omp_get_max_active_levels** routine returns the value of the *max-active-levels-var* ICV, which determines the maximum number of nested active parallel regions when the innermost parallel region is generated by the current task.

Format

C / C++

```
int omp_get_max_active_levels(void);
```

C / C++

Fortran

```
integer function omp_get_max_active_levels()
```

Fortran

Binding

The binding task set for an **omp_get_max_active_levels** region is the generating task.

Effect

The **omp_get_max_active_levels** routine returns the value of the *max-active-levels-var* ICV. The current task may only generate an active parallel region if the returned value is greater than the value of the *active-levels-var* ICV.

Cross References

- *max-active-levels-var* ICV, see Section 2.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 21.1.4.
- **parallel** construct, see Section 10.1.
- **omp_get_supported_active_levels** routine, see Section 18.2.14.
- **omp_set_max_active_levels** routine, see Section 18.2.15.

18.2.17 omp_get_level

Summary

The **omp_get_level** routine returns the value of the *levels-var* ICV.

Format

C / C++

```
int omp_get_level(void);
```

C / C++

Fortran

```
integer function omp_get_level()
```

Fortran

Binding

The binding task set for an **omp_get_level** region is the generating task.

Effect
The effect of the **omp_get_level** routine is to return the number of nested **parallel** regions (whether active or inactive) that enclose the current task such that all of the **parallel** regions are enclosed by the outermost initial task region on the current device.

Cross References
- *levels-var* ICV, see Section 2.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 21.1.4.
- **parallel** construct, see Section 10.1.
- **omp_get_active_level** routine, see Section 18.2.20.

18.2.18 omp_get_ancestor_thread_num

Summary
The **omp_get_ancestor_thread_num** routine returns, for a given nested level of the current thread, the thread number of the ancestor of the current thread.

Format
C / C++

```
int omp_get_ancestor_thread_num(int level);
```

C / C++

Fortran

```
integer function omp_get_ancestor_thread_num(level)
integer level
```

Fortran

Binding
The binding thread set for an **omp_get_ancestor_thread_num** region is the encountering thread. The binding region for an **omp_get_ancestor_thread_num** region is the innermost enclosing **parallel** region.

Effect
The **omp_get_ancestor_thread_num** routine returns the thread number of the ancestor at a given nest level of the current thread or the thread number of the current thread. If the requested nest level is outside the range of 0 and the nest level of the current thread, as returned by the **omp_get_level** routine, the routine returns -1.

Note – When the **omp_get_ancestor_thread_num** routine is called with a value of **level**=0, the routine always returns 0. If **level=omp_get_level()**, the routine has the same effect as the **omp_get_thread_num** routine.

Cross References

- **parallel** construct, see Section 10.1.
- **omp_get_level** routine, see Section 18.2.17.
- **omp_get_num_threads** routine, see Section 18.2.2.
- **omp_get_team_size** routine, see Section 18.2.19.
- **omp_get_thread_num** routine, see Section 18.2.4.

18.2.19 omp_get_team_size

Summary

The **omp_get_team_size** routine returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.

Format

C / C++

```
int omp_get_team_size(int level);
```

C / C++

Fortran

```
integer function omp_get_team_size(level)
integer level
```

Fortran

Binding

The binding thread set for an **omp_get_team_size** region is the encountering thread. The binding region for an **omp_get_team_size** region is the innermost enclosing **parallel** region.

Effect

The **omp_get_team_size** routine returns the size of the thread team to which the ancestor or the current thread belongs. If the requested nested level is outside the range of 0 and the nested level of the current thread, as returned by the **omp_get_level** routine, the routine returns -1. Inactive parallel regions are regarded like active parallel regions executed with one thread.

Note – When the **omp_get_team_size** routine is called with a value of **level**=0, the routine always returns 1. If **level=omp_get_level()**, the routine has the same effect as the **omp_get_num_threads** routine.

Cross References

- **omp_get_ancestor_thread_num** routine, see Section 18.2.18.
- **omp_get_level** routine, see Section 18.2.17.
- **omp_get_num_threads** routine, see Section 18.2.2.

18.2.20 omp_get_active_level

Summary

The **omp_get_active_level** routine returns the value of the *active-level-var* ICV.

Format

C / C++

```
int omp_get_active_level(void);
```

C / C++

Fortran

```
integer function omp_get_active_level()
```

Fortran

Binding

The binding task set for the an **omp_get_active_level** region is the generating task.

Effect

The effect of the **omp_get_active_level** routine is to return the number of nested active **parallel** regions enclosing the current task such that all of the **parallel** regions are enclosed by the outermost initial task region on the current device.

Cross References

- *active-levels-var* ICV, see Section 2.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 21.1.4.
- **omp_get_level** routine, see Section 18.2.17.
- **omp_get_max_active_levels** routine, see Section 18.2.16.
- **omp_set_max_active_levels** routine, see Section 18.2.15.

18.3 Thread Affinity Routines

This section describes routines that affect and access thread affinity policies that are in effect.

18.3.1 omp_get_proc_bind

Summary

The **omp_get_proc_bind** routine returns the thread affinity policy to be used for the subsequent nested **parallel** regions that do not specify a **proc_bind** clause.

Format

C / C++

```
omp_proc_bind_t omp_get_proc_bind(void);
```

C / C++

Fortran

```
integer (kind=omp_proc_bind_kind) function omp_get_proc_bind()
```

Fortran

Constraints on Arguments

The value returned by this routine must be one of the valid affinity policy kinds. The C/C++ header file (**omp.h**) and the Fortran include file (**omp_lib.h**) and/or Fortran module file (**omp_lib**) define the valid constants. The valid constants must include the following:

C / C++

```
typedef enum omp_proc_bind_t {
  omp_proc_bind_false = 0,
  omp_proc_bind_true = 1,
  omp_proc_bind_primary = 2,
  omp_proc_bind_master = omp_proc_bind_primary, // (deprecated)
  omp_proc_bind_close = 3,
  omp_proc_bind_spread = 4
} omp_proc_bind_t;
```

C / C++

Fortran

```fortran
integer (kind=omp_proc_bind_kind), &
                parameter :: omp_proc_bind_false = 0
integer (kind=omp_proc_bind_kind), &
                parameter :: omp_proc_bind_true = 1
integer (kind=omp_proc_bind_kind), &
                parameter :: omp_proc_bind_primary = 2
integer (kind=omp_proc_bind_kind), &
                parameter :: omp_proc_bind_master = &
                omp_proc_bind_primary         !   (deprecated)
integer (kind=omp_proc_bind_kind), &
                parameter :: omp_proc_bind_close = 3
integer (kind=omp_proc_bind_kind), &
                parameter :: omp_proc_bind_spread = 4
```

Fortran

Binding

The binding task set for an **omp_get_proc_bind** region is the generating task.

Effect

The effect of this routine is to return the value of the first element of the *bind-var* ICV of the current task. See Section 10.1.3 for the rules that govern the thread affinity policy.

Cross References

- *bind-var* ICV, see Section 2.
- **OMP_PLACES** environment variable, see Section 21.1.6.
- **OMP_PROC_BIND** environment variable, see Section 21.1.7.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **omp_get_num_places** routine, see Section 18.3.2.

18.3.2 omp_get_num_places

Summary

The **omp_get_num_places** routine returns the number of places available to the execution environment in the place list.

Format

C / C++

```
int omp_get_num_places(void);
```

C / C++

Fortran

```
integer function omp_get_num_places()
```

Fortran

Binding

The binding thread set for an **omp_get_num_places** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **omp_get_num_places** routine returns the number of places in the place list. This value is equivalent to the number of places in the *place-partition-var* ICV in the execution environment of the initial task.

Cross References

- *place-partition-var* ICV, see Section 2.
- **OMP_PLACES** environment variable, see Section 21.1.6.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **omp_get_place_num** routine, see Section 18.3.5.

18.3.3 omp_get_place_num_procs

Summary

The **omp_get_place_num_procs** routine returns the number of processors available to the execution environment in the specified place.

Format

C / C++

```
int omp_get_place_num_procs(int place_num);
```

C / C++

Fortran

```
integer function omp_get_place_num_procs(place_num)
integer place_num
```

Fortran

Binding

The binding thread set for an **omp_get_place_num_procs** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **omp_get_place_num_procs** routine returns the number of processors associated with the place numbered *place_num*. The routine returns zero when *place_num* is negative, or is greater than or equal to the value returned by **omp_get_num_places()**.

Cross References

- *place-partition-var* ICV, see Section 2.
- **OMP_PLACES** environment variable, see Section 21.1.6.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **omp_get_num_places** routine, see Section 18.3.2.
- **omp_get_place_proc_ids** routine, see Section 18.3.4.

18.3.4 omp_get_place_proc_ids

Summary

The **omp_get_place_proc_ids** routine returns the numerical identifiers of the processors available to the execution environment in the specified place.

Format

C / C++

```
void omp_get_place_proc_ids(int place_num, int *ids);
```

C / C++

Fortran

```
subroutine omp_get_place_proc_ids(place_num, ids)
integer place_num
integer ids(*)
```

Fortran

Binding

The binding thread set for an **omp_get_place_proc_ids** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **omp_get_place_proc_ids** routine returns the numerical identifiers of each processor associated with the place numbered *place_num*. The numerical identifiers are non-negative and their meaning is implementation defined. The numerical identifiers are returned in the array *ids* and their order in the array is implementation defined. The array must be sufficiently large to contain **omp_get_place_num_procs(***place_num***)** integers; otherwise, the behavior is unspecified. The routine has no effect when *place_num* has a negative value or a value greater than or equal to **omp_get_num_places()**.

Cross References

- *place-partition-var* ICV, see Section 2.
- **OMP_PLACES** environment variable, see Section 21.1.6.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **omp_get_num_places** routine, see Section 18.3.2.
- **omp_get_place_num_procs** routine, see Section 18.3.3.

18.3.5 omp_get_place_num

Summary

The **omp_get_place_num** routine returns the place number of the place to which the encountering thread is bound.

Format

C / C++

```
int omp_get_place_num(void);
```

C / C++

Fortran

```
integer function omp_get_place_num()
```

Fortran

Binding

The binding thread set for an **omp_get_place_num** region is the encountering thread.

Effect

When the encountering thread is bound to a place, the **omp_get_place_num** routine returns the place number associated with the thread. The returned value is between 0 and one less than the value returned by **omp_get_num_places()**, inclusive. When the encountering thread is not bound to a place, the routine returns -1.

Cross References

- *place-partition-var* ICV, see Section 2.
- **OMP_PLACES** environment variable, see Section 21.1.6.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **omp_get_num_places** routine, see Section 18.3.2.

18.3.6 omp_get_partition_num_places

Summary

The **omp_get_partition_num_places** routine returns the number of places in the place partition of the innermost implicit task.

Format

C / C++

```
int omp_get_partition_num_places(void);
```

C / C++

Fortran

```
integer function omp_get_partition_num_places()
```

Fortran

Binding

The binding task set for an **omp_get_partition_num_places** region is the encountering implicit task.

Effect

The **omp_get_partition_num_places** routine returns the number of places in the *place-partition-var* ICV.

Cross References

- *place-partition-var* ICV, see Section 2.
- **OMP_PLACES** environment variable, see Section 21.1.6.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **omp_get_num_places** routine, see Section 18.3.2.

18.3.7 omp_get_partition_place_nums

Summary

The **omp_get_partition_place_nums** routine returns the list of place numbers corresponding to the places in the *place-partition-var* ICV of the innermost implicit task.

Format

C / C++

```
void omp_get_partition_place_nums(int *place_nums);
```

C / C++

Fortran

```
subroutine omp_get_partition_place_nums(place_nums)
integer place_nums(*)
```

Fortran

Binding

The binding task set for an **omp_get_partition_place_nums** region is the encountering implicit task.

Effect

The **omp_get_partition_place_nums** routine returns the list of place numbers that correspond to the places in the *place-partition-var* ICV of the innermost implicit task. The array must be sufficiently large to contain **omp_get_partition_num_places()** integers; otherwise, the behavior is unspecified.

Cross References

- *place-partition-var* ICV, see Section 2.
- **OMP_PLACES** environment variable, see Section 21.1.6.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **omp_get_partition_num_places** routine, see Section 18.3.6.

18.3.8 omp_set_affinity_format

Summary

The **omp_set_affinity_format** routine sets the affinity format to be used on the device by setting the value of the *affinity-format-var* ICV.

Format

C / C++

```
void omp_set_affinity_format(const char *format);
```

C / C++

Fortran

```
subroutine omp_set_affinity_format(format)
character(len=*),intent(in) :: format
```

Fortran

Binding

When called from a sequential part of the program, the binding thread set for an **omp_set_affinity_format** region is the encountering thread. When called from within any **parallel** or **teams** region, the binding thread set (and binding region, if required) for the **omp_set_affinity_format** region is implementation defined.

Effect

The effect of **omp_set_affinity_format** routine is to copy the character string specified by the *format* argument into the *affinity-format-var* ICV on the current device.

This routine has the described effect only when called from a sequential part of the program. When called from within a **parallel** or **teams** region, the effect of this routine is implementation defined.

Cross References

- **OMP_AFFINITY_FORMAT** environment variable, see Section 21.2.5.
- **OMP_DISPLAY_AFFINITY** environment variable, see Section 21.2.4.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **omp_capture_affinity** routine, see Section 18.3.11.
- **omp_display_affinity** routine, see Section 18.3.10.
- **omp_get_affinity_format** routine, see Section 18.3.9.

18.3.9 omp_get_affinity_format

Summary

The **omp_get_affinity_format** routine returns the value of the *affinity-format-var* ICV on the device.

Format

C / C++

```
size_t omp_get_affinity_format(char *buffer, size_t size);
```

C / C++

Fortran

```
integer function omp_get_affinity_format(buffer)
character(len=*),intent(out) :: buffer
```

Fortran

Binding

When called from a sequential part of the program, the binding thread set for an **omp_get_affinity_format** region is the encountering thread. When called from within any **parallel** or **teams** region, the binding thread set (and binding region, if required) for the **omp_get_affinity_format** region is implementation defined.

Effect

C / C++

The **omp_get_affinity_format** routine returns the number of characters in the *affinity-format-var* ICV on the current device, excluding the terminating null byte (**'\0'**) and if *size* is non-zero, writes the value of the *affinity-format-var* ICV on the current device to *buffer* followed by a null byte. If the return value is larger or equal to *size*, the affinity format specification is truncated, with the terminating null byte stored to ***buffer*****[*****size*****-1]**. If *size* is zero, nothing is stored and *buffer* may be *NULL*.

C / C++

Fortran

The **omp_get_affinity_format** routine returns the number of characters that are required to hold the *affinity-format-var* ICV on the current device and writes the value of the *affinity-format-var* ICV on the current device to *buffer*. If the return value is larger than **len(*****buffer*****)**, the affinity format specification is truncated.

Fortran

If the *buffer* argument does not conform to the specified format then the result is implementation defined.

Cross References

- **OMP_AFFINITY_FORMAT** environment variable, see Section 21.2.5.
- **OMP_DISPLAY_AFFINITY** environment variable, see Section 21.2.4.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **omp_capture_affinity** routine, see Section 18.3.11.
- **omp_display_affinity** routine, see Section 18.3.10.
- **omp_set_affinity_format** routine, see Section 18.3.8.

18.3.10 omp_display_affinity

Summary

The **omp_display_affinity** routine prints the OpenMP thread affinity information using the format specification provided.

Format

C / C++

```
void omp_display_affinity(const char *format);
```

C / C++

Fortran

```
subroutine omp_display_affinity(format)
character(len=*),intent(in) :: format
```

Fortran

Binding

The binding thread set for an **omp_display_affinity** region is the encountering thread.

Effect

The **omp_display_affinity** routine prints the thread affinity information of the current thread in the format specified by the *format* argument, followed by a *new-line*. If the *format* is *NULL* (for C/C++) or a zero-length string (for Fortran and C/C++), the value of the *affinity-format-var* ICV is used. If the *format* argument does not conform to the specified format then the result is implementation defined.

Cross References

- **OMP_AFFINITY_FORMAT** environment variable, see Section 21.2.5.
- **OMP_DISPLAY_AFFINITY** environment variable, see Section 21.2.4.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **omp_capture_affinity** routine, see Section 18.3.11.
- **omp_get_affinity_format** routine, see Section 18.3.9.
- **omp_set_affinity_format** routine, see Section 18.3.8.

18.3.11 omp_capture_affinity

Summary

The **omp_capture_affinity** routine prints the OpenMP thread affinity information into a buffer using the format specification provided.

Format

C / C++

```
size_t omp_capture_affinity(
  char *buffer,
  size_t size,
  const char *format
);
```

C / C++

Fortran

```
integer function omp_capture_affinity(buffer,format)
character(len=*),intent(out) :: buffer
character(len=*),intent(in) :: format
```

Fortran

Binding

The binding thread set for an **omp_capture_affinity** region is the encountering thread.

Effect

C / C++

The **omp_capture_affinity** routine returns the number of characters in the entire thread affinity information string excluding the terminating null byte (**'\0'**) and if *size* is non-zero, writes the thread affinity information of the current thread in the format specified by the *format* argument into the character string **buffer** followed by a null byte. If the return value is larger or equal to *size*, the thread affinity information string is truncated, with the terminating null byte stored to ***buffer*[*size*-1]**. If *size* is zero, nothing is stored and *buffer* may be *NULL*. If the *format* is *NULL* or a zero-length string, the value of the *affinity-format-var* ICV is used.

C / C++

Fortran

The **omp_capture_affinity** routine returns the number of characters required to hold the entire thread affinity information string and prints the thread affinity information of the current thread into the character string **buffer** with the size of **len(*buffer*)** in the format specified by the *format* argument. If the *format* is a zero-length string, the value of the *affinity-format-var* ICV is used. If the return value is larger than **len(*buffer*)**, the thread affinity information string is truncated. If the *format* is a zero-length string, the value of the *affinity-format-var* ICV is used.

Fortran

If the *format* argument does not conform to the specified format then the result is implementation defined.

Cross References

- **OMP_AFFINITY_FORMAT** environment variable, see Section 21.2.5.
- **OMP_DISPLAY_AFFINITY** environment variable, see Section 21.2.4.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **omp_display_affinity** routine, see Section 18.3.10.
- **omp_get_affinity_format** routine, see Section 18.3.9.
- **omp_set_affinity_format** routine, see Section 18.3.8.

18.4 Teams Region Routines

This section describes routines that affect and monitor the league of teams that may execute a **teams** region.

18.4.1 omp_get_num_teams

Summary

The **omp_get_num_teams** routine returns the number of initial teams in the current **teams** region.

Format

C / C++

```
int omp_get_num_teams(void);
```

C / C++

Fortran

```
integer function omp_get_num_teams()
```

Fortran

Binding

The binding task set for an **omp_get_num_teams** region is the generating task

Effect

The effect of this routine is to return the number of initial teams in the current **teams** region. The routine returns 1 if it is called from outside of a **teams** region.

Cross References

- **teams** construct, see Section 10.2.
- **omp_get_team_num** routine, see Section 18.4.2.

18.4.2 `omp_get_team_num`

Summary

The **omp_get_team_num** routine returns the initial team number of the calling thread.

Format

C / C++

```
int omp_get_team_num(void);
```

C / C++

Fortran

```
integer function omp_get_team_num()
```

Fortran

Binding

The binding task set for an **omp_get_team_num** region is the generating task.

Effect

The **omp_get_team_num** routine returns the initial team number of the calling thread. The initial team number is an integer between 0 and one less than the value returned by **omp_get_num_teams()**, inclusive. The routine returns 0 if it is called outside of a **teams** region.

Cross References

- **teams** construct, see Section 10.2.
- **omp_get_num_teams** routine, see Section 18.4.1.

18.4.3 `omp_set_num_teams`

Summary

The **omp_set_num_teams** routine affects the number of threads to be used for subsequent **teams** regions that do not specify a **num_teams** clause, by setting the value of the *nteams-var* ICV of the current task.

Format

C / C++

```
void omp_set_num_teams(int num_teams);
```

C / C++

Fortran

```
subroutine omp_set_num_teams(num_teams)
integer num_teams
```

Fortran

Constraints on Arguments

The value of the argument passed to this routine must evaluate to a positive integer, or else the behavior of this routine is implementation defined.

Binding

The binding task set for an **omp_set_num_teams** region is the generating task.

Effect

The effect of this routine is to set the value of the *nteams-var* ICV of the current task to the value specified in the argument.

Restrictions

Restrictions to the **omp_set_num_teams** routine are as follows:

- The routine may not be called from within a parallel region that is not the implicit parallel region that surrounds the whole OpenMP program.

Cross References

- *nteams-var* ICV, see Section 2.
- **OMP_NUM_TEAMS** environment variable, see Section 21.6.1.
- **teams** construct and **num_teams** clause, see Section 10.2.
- **omp_get_max_teams** routine, see Section 18.4.4.
- **omp_get_num_teams** routine, see Section 18.4.1.

18.4.4 omp_get_max_teams

Summary

The **omp_get_max_teams** routine returns an upper bound on the number of teams that could be created by a **teams** construct without a **num_teams** clause that is encountered after execution returns from this routine.

Format

C / C++

```
int omp_get_max_teams(void);
```

C / C++

Fortran

```
integer function omp_get_max_teams()
```

Fortran

Binding

The binding task set for an **omp_get_max_teams** region is the generating task.

Effect

The value returned by **omp_get_max_teams** is the value of the *nteams-var* ICV of the current task. This value is also an upper bound on the number of teams that can be created by a **teams** construct without a **num_teams** clause that is encountered after execution returns from this routine.

Cross References

- *nteams-var* ICV, see Section 2.
- **teams** construct and **num_teams** clause, see Section 10.2.
- **omp_get_num_teams** routine, see Section 18.4.1.
- **omp_set_num_teams** routine, see Section 18.4.3.

18.4.5 omp_set_teams_thread_limit

Summary

The **omp_set_teams_thread_limit** routine defines the maximum number of OpenMP threads that can participate in each contention group created by a **teams** construct.

Format

C / C++

```
void omp_set_teams_thread_limit(int thread_limit);
```

C / C++

Fortran

```
subroutine omp_set_teams_thread_limit(thread_limit)
integer thread_limit
```

Fortran

Constraints on Arguments

The value of the argument passed to this routine must evaluate to a positive integer, or else the behavior of this routine is implementation defined.

Binding

The binding task set for an **omp_set_teams_thread_limit** region is the generating task.

Effect

The **omp_set_teams_thread_limit** routine sets the value of the *teams-thread-limit-var* ICV to the value of the *thread_limit* argument.

If the value of *thread_limit* exceeds the number of OpenMP threads that an implementation supports for each contention group created by a **teams** construct, the value of the *teams-thread-limit-var* ICV will be set to the number that is supported by the implementation.

Restrictions

Restrictions to the **omp_set_teams_thread_limit** routine are as follows:

- The routine may not be called from within a parallel region other than the implicit parallel region that surrounds the whole OpenMP program.

Cross References

- *teams_thread-limit-var* ICV, see Section 2.
- **OMP_TEAMS_THREAD_LIMIT** environment variable, see Section 21.6.2.
- **teams** construct and **thread_limit** clause, see Section 10.2.
- **omp_get_teams_thread_limit** routine, see Section 18.4.6.

18.4.6 omp_get_teams_thread_limit

Summary

The **omp_get_teams_thread_limit** routine returns the maximum number of OpenMP threads available to participate in each contention group created by a **teams** construct.

Format

C / C++

```c
int omp_get_teams_thread_limit(void);
```

C / C++

Fortran

```fortran
integer function omp_get_teams_thread_limit()
```

Fortran

Binding

The binding task set for an **omp_get_teams_thread_limit** region is the generating task.

Effect

The **omp_get_teams_thread_limit** routine returns the value of the *teams-thread-limit-var* ICV.

Cross References

- *teams_thread-limit-var* ICV, see Section 2.
- **OMP_TEAMS_THREAD_LIMIT** environment variable, see Section 21.6.2.
- **teams** construct and **thread_limit** clause, see Section 10.2.
- **omp_set_teams_thread_limit** routine, see Section 18.4.5.

18.5 Tasking Routines

This section describes routines that pertain to OpenMP explicit tasks.

18.5.1 omp_get_max_task_priority

Summary

The **omp_get_max_task_priority** routine returns the maximum value that can be specified in the **priority** clause.

Format

C / C++

```
int omp_get_max_task_priority(void);
```

C / C++

Fortran

```
integer function omp_get_max_task_priority()
```

Fortran

Binding

The binding thread set for an **omp_get_max_task_priority** region is all threads on the device. The effect of executing this routine is not related to any specific region that corresponds to any construct or API routine.

Effect

The **omp_get_max_task_priority** routine returns the value of the *max-task-priority-var* ICV, which determines the maximum value that can be specified in the **priority** clause.

Cross References

- *max-task-priority-var*, see Section 2.
- **task** construct, see Section 12.5.

18.5.2 `omp_in_final`

Summary

The **`omp_in_final`** routine returns *true* if the routine is executed in a final task region; otherwise, it returns *false*.

Format

C / C++

```
int omp_in_final(void);
```

C / C++

Fortran

```
logical function omp_in_final()
```

Fortran

Binding

The binding task set for an **`omp_in_final`** region is the generating task.

Effect

`omp_in_final` returns *true* if the enclosing task region is final. Otherwise, it returns *false*.

Cross References

- **`task`** construct, see Section 12.5.

18.6 Resource Relinquishing Routines

This section describes routines that relinquish resources used by the OpenMP runtime.

18.6.1 `omp_pause_resource`

Summary

The **`omp_pause_resource`** routine allows the runtime to relinquish resources used by OpenMP on the specified device.

Format

C / C++

```
int omp_pause_resource(
  omp_pause_resource_t kind,
  int device_num
);
```

C / C++

Fortran

```
integer function omp_pause_resource(kind, device_num)
integer (kind=omp_pause_resource_kind) kind
integer device_num
```

Fortran

Constraints on Arguments

The first argument passed to this routine can be one of the valid OpenMP pause kind, or any implementation specific pause kind. The C/C++ header file (**omp.h**) and the Fortran include file (**omp_lib.h**) and/or Fortran module file (**omp_lib**) define the valid constants. The valid constants must include the following, which can be extended with implementation-specific values:

Format

C / C++

```
typedef enum omp_pause_resource_t {
  omp_pause_soft = 1,
  omp_pause_hard = 2
} omp_pause_resource_t;
```

C / C++

Fortran

```
integer (kind=omp_pause_resource_kind), parameter :: &
  omp_pause_soft = 1
integer (kind=omp_pause_resource_kind), parameter :: &
  omp_pause_hard = 2
```

Fortran

The second argument passed to this routine indicates the device that will be paused. The **device_num** parameter must be a conforming device number. If the device number has the value **omp_invalid_device**, runtime error termination is performed.

Binding

The binding task set for an **omp_pause_resource** region is the whole program.

Effect

The **omp_pause_resource** routine allows the runtime to relinquish resources used by OpenMP on the specified device.

If successful, the **omp_pause_hard** value results in a hard pause for which the OpenMP state is not guaranteed to persist across the **omp_pause_resource** call. A hard pause may relinquish any data allocated by OpenMP on a given device, including data allocated by memory routines for that device as well as data present on the device as a result of a declare target directive or **target data** construct. A hard pause may also relinquish any data associated with a **threadprivate** directive. When relinquished and when applicable, base language appropriate deallocation/finalization is performed. When relinquished and when applicable, mapped data on a device will not be copied back from the device to the host.

If successful, the **omp_pause_soft** value results in a soft pause for which the OpenMP state is guaranteed to persist across the call, with the exception of any data associated with a **threadprivate** directive, which may be relinquished across the call. When relinquished and when applicable, base language appropriate deallocation/finalization is performed.

Note – A hard pause may relinquish more resources, but may resume processing OpenMP regions more slowly. A soft pause allows OpenMP regions to restart more quickly, but may relinquish fewer resources. An OpenMP implementation will reclaim resources as needed for OpenMP regions encountered after the **omp_pause_resource** region. Since a hard pause may unmap data on the specified device, appropriate data mapping is required before using data on the specified device after the **omp_pause_region** region.

The routine returns zero in case of success, and non-zero otherwise.

Tool Callbacks

If the tool is not allowed to interact with the specified device after encountering this call, then the runtime must call the tool finalizer for that device.

Restrictions

Restrictions to the **omp_pause_resource** routine are as follows:

- The **omp_pause_resource** region may not be nested in any explicit OpenMP region.
- The routine may only be called when all explicit tasks have finalized execution.

Cross References

- Declare target directive, see Section 7.8.
- To pause resources on all devices at once, see Section 18.6.2.
- **target** construct, see Section 13.8.
- **threadprivate** directives, see Section 5.2.

18.6.2 omp_pause_resource_all

Summary

The **omp_pause_resource_all** routine allows the runtime to relinquish resources used by OpenMP on all devices.

Format

C / C++

```
int omp_pause_resource_all(omp_pause_resource_t kind);
```

C / C++

Fortran

```
integer function omp_pause_resource_all(kind)
integer (kind=omp_pause_resource_kind) kind
```

Fortran

Binding

The binding task set for an **omp_pause_resource_all** region is the whole program.

Effect

The **omp_pause_resource_all** routine allows the runtime to relinquish resources used by OpenMP on all devices. It is equivalent to calling the **omp_pause_resource** routine once for each available device, including the host device.

The argument *kind* passed to this routine can be one of the valid OpenMP pause kind as defined in Section 18.6.1, or any implementation-specific pause kind.

Tool Callbacks

If the tool is not allowed to interact with a given device after encountering this call, then the runtime must call the tool finalizer for that device.

Restrictions

Restrictions to the **omp_pause_resource_all** routine are as follows:

- The **omp_pause_resource_all** region may not be nested in any explicit OpenMP region.
- The routine may only be called when all explicit tasks have finalized execution.

Cross References

- Declare target directive, see Section 7.8.
- To pause resources on a specific device only, see Section 18.6.1.
- **target** construct, see Section 13.8.

18.7 Device Information Routines

This section describes routines that pertain to the set of devices that are accessible to an OpenMP program.

18.7.1 `omp_get_num_procs`

Summary

The **`omp_get_num_procs`** routine returns the number of processors available to the device.

Format

C / C++

```
int omp_get_num_procs(void);
```

C / C++

Fortran

```
integer function omp_get_num_procs()
```

Fortran

Binding

The binding thread set for an **`omp_get_num_procs`** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **`omp_get_num_procs`** routine returns the number of processors that are available to the device at the time the routine is called. This value may change between the time that it is determined by the **`omp_get_num_procs`** routine and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation.

Cross References

- **`omp_get_num_places`** routine, see Section 18.3.2.
- **`omp_get_place_num`** routine, see Section 18.3.5.
- **`omp_get_place_num_procs`** routine, see Section 18.3.3.
- **`omp_get_place_proc_ids`** routine, see Section 18.3.4.

18.7.2 `omp_set_default_device`

Summary

The **`omp_set_default_device`** routine controls the default target device by assigning the value of the *default-device-var* ICV.

Format

C / C++

```
void omp_set_default_device(int device_num);
```

C / C++

Fortran

```
subroutine omp_set_default_device(device_num)
integer device_num
```

Fortran

Binding

The binding task set for an **omp_set_default_device** region is the generating task.

Effect

The effect of this routine is to set the value of the *default-device-var* ICV of the current task to the value specified in the argument. When called from within a **target** region the effect of this routine is unspecified.

Cross References

- *default-device-var*, see Section 2.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 21.2.7.
- **omp_get_default_device**, see Section 18.7.3.
- **target** construct, see Section 13.8.

18.7.3 omp_get_default_device

Summary

The **omp_get_default_device** routine returns the default target device.

Format

C / C++

```
int omp_get_default_device(void);
```

C / C++

Fortran

```
integer function omp_get_default_device()
```

Fortran

Binding

The binding task set for an **omp_get_default_device** region is the generating task.

Effect

The **omp_get_default_device** routine returns the value of the *default-device-var* ICV of the current task. When called from within a **target** region the effect of this routine is unspecified.

Cross References

- *default-device-var*, see Section 2.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 21.2.7.
- **omp_set_default_device**, see Section 18.7.2.
- **target** construct, see Section 13.8.

18.7.4 omp_get_num_devices

Summary

The **omp_get_num_devices** routine returns the number of non-host devices available for offloading code or data.

Format

C / C++

```
int omp_get_num_devices(void);
```

C / C++

Fortran

```
integer function omp_get_num_devices()
```

Fortran

Binding

The binding task set for an **omp_get_num_devices** region is the generating task.

Effect

The **omp_get_num_devices** routine returns the number of available non-host devices onto which code or data may be offloaded. When called from within a **target** region the effect of this routine is unspecified.

Cross References

- **omp_get_default_device**, see Section 18.7.3.
- **omp_get_device_num**, see Section 18.7.5.
- **target** construct, see Section 13.8.

18.7.5 omp_get_device_num

Summary

The **omp_get_device_num** routine returns the device number of the device on which the calling thread is executing.

Format

C / C++

```
int omp_get_device_num(void);
```

C / C++

Fortran

```
integer function omp_get_device_num()
```

Fortran

Binding

The binding task set for an **omp_get_device_num** region is the generating task.

Effect

The **omp_get_device_num** routine returns the device number of the device on which the calling thread is executing. When called on the host device, it will return the same value as the **omp_get_initial_device** routine.

Cross References

- **omp_get_default_device**, see Section 18.7.3.
- **omp_get_initial_device** routine, see Section 18.7.7.
- **omp_get_num_devices**, see Section 18.7.4.
- **target** construct, see Section 13.8.

18.7.6 omp_is_initial_device

Summary

The **omp_is_initial_device** routine returns *true* if the current task is executing on the host device; otherwise, it returns *false*.

Format

C / C++

```
int omp_is_initial_device(void);
```

C / C++

Fortran

```
logical function omp_is_initial_device()
```

Fortran

Binding

The binding task set for an **omp_is_initial_device** region is the generating task.

Effect

The effect of this routine is to return *true* if the current task is executing on the host device; otherwise, it returns *false*.

Cross References

- Device memory routines, see Section 18.8.
- **omp_get_initial_device** routine, see Section 18.7.7.

18.7.7 omp_get_initial_device

Summary

The **omp_get_initial_device** routine returns a device number that represents the host device.

Format

C / C++

```
int omp_get_initial_device(void);
```

C / C++

Fortran

```
integer function omp_get_initial_device()
```

Fortran

Binding

The binding task set for an **omp_get_initial_device** region is the generating task.

Effect

The effect of this routine is to return the device number of the host device. The value of the device number is the value returned by the **omp_get_num_devices** routine. When called from within a **target** region the effect of this routine is unspecified.

Cross References

- Device memory routines, see Section 18.8.
- **omp_is_initial_device** routine, see Section 18.7.6.
- **target** construct, see Section 13.8.

18.8 Device Memory Routines

This section describes routines that support allocation of memory and management of pointers in the data environments of target devices.

If the *device_num*, *src_device_num*, or *dst_device_num* argument of a device memory routine has the value **omp_invalid_device**, runtime error termination is performed.

18.8.1 omp_target_alloc

Summary

The **omp_target_alloc** routine allocates memory in a device data environment and returns a device pointer to that memory.

Format

C / C++

```
void* omp_target_alloc(size_t size, int device_num);
```

C / C++

Fortran

```
type(c_ptr) function omp_target_alloc(size, device_num) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t, c_int
integer(c_size_t), value :: size
integer(c_int), value  :: device_num
```

Fortran

Constraints on Arguments

The *device_num* argument must be a conforming device number.

Binding

The binding task set for an **omp_target_alloc** region is the generating task, which is the *target task* generated by the call to the **omp_target_alloc** routine.

Effect

The **omp_target_alloc** routine returns a device pointer that references the device address of a storage location of *size* bytes. The storage location is dynamically allocated in the device data environment of the device specified by *device_num*.

The **omp_target_alloc** routine executes as if part of a target task that is generated by the call to the routine and that is an included task.

The **omp_target_alloc** routine returns *NULL* if it cannot dynamically allocate the memory in the device data environment.

The device pointer returned by **omp_target_alloc** can be used in an **is_device_ptr** clause, Section 13.8.

Fortran

The **omp_target_alloc** routine requires an explicit interface and so might not be provided in **omp_lib.h**.

Fortran

Execution Model Events

The *target-data-allocation-begin* event occurs before a thread initiates a data allocation on a target device.

The *target-data-allocation-end* event occurs after a thread initiates a data allocation on a target device.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_begin** as its endpoint argument for each occurrence of a *target-data-allocation-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_end** as its endpoint argument for each occurrence of a *target-data-allocation-end* event in that thread. These callbacks have type signature **ompt_callback_target_data_op_emi_t**.

A thread dispatches a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-allocation-end* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**.

Restrictions

Restrictions to the **omp_target_alloc** routine are as follows.

- Freeing the storage returned by **omp_target_alloc** with any routine other than **omp_target_free** results in unspecified behavior.
- When called from within a **target** region the effect is unspecified.

C / C++

- Unless the **unified_address** clause appears on a **requires** directive in the compilation unit, pointer arithmetic is not supported on the device pointer returned by **omp_target_alloc**.

C / C++

Cross References

- **ompt_callback_target_data_op_t** or **ompt_callback_target_data_op_emi_t** callback type, see Section 19.5.2.25.
- **omp_target_free** routine, see Section 18.8.2.
- **target** construct, see Section 13.8.

18.8.2 omp_target_free

Summary

The **omp_target_free** routine frees the device memory allocated by the **omp_target_alloc** routine.

Format

C / C++

```
void omp_target_free(void *device_ptr, int device_num);
```

C / C++

Fortran

```
subroutine omp_target_free(device_ptr, device_num) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_int
type(c_ptr), value :: device_ptr
integer(c_int), value :: device_num
```

Fortran

Constraints on Arguments

A program that calls **omp_target_free** with a non-null pointer that does not have a value returned from **omp_target_alloc** is non-conforming. The *device_num* argument must be a conforming device number.

Binding

The binding task set for an **omp_target_free** region is the generating task, which is the *target task* generated by the call to the **omp_target_free** routine.

Effect

The **omp_target_free** routine frees the memory in the device data environment associated with *device_ptr*. If *device_ptr* is *NULL*, the operation is ignored.

The **omp_target_free** routine executes as if part of a target task that is generated by the call to the routine and that is an included task.

Synchronization must be inserted to ensure that all accesses to *device_ptr* are completed before the call to **omp_target_free**.

Fortran

The **omp_target_free** routine requires an explicit interface and so might not be provided in **omp_lib.h**.

Fortran

Execution Model Events

The *target-data-free-begin* event occurs before a thread initiates a data free on a target device.

The *target-data-free-end* event occurs after a thread initiates a data free on a target device.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_begin** as its endpoint argument for each occurrence of a *target-data-free-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_end** as its endpoint argument for each occurrence of a *target-data-free-end* event in that thread. These callbacks have type signature **ompt_callback_target_data_op_emi_t**.

A thread dispatches a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-free-begin* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**.

Restrictions

Restrictions to the **omp_target_free** routine are as follows.

- When called from within a **target** region the effect is unspecified.

Cross References

- **ompt_callback_target_data_op_t** or **ompt_callback_target_data_op_emi_t** callback type, see Section 19.5.2.25.
- **omp_target_alloc** routine, see Section 18.8.1.
- **target** construct, see Section 13.8.

18.8.3 omp_target_is_present

Summary

The **omp_target_is_present** routine tests whether a host pointer refers to storage that is mapped to a given device.

Format

C / C++

```
int omp_target_is_present(const void *ptr, int device_num);
```

C / C++

Fortran

```
integer(c_int) function omp_target_is_present(ptr, device_num) &
  bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_int
type(c_ptr), value :: ptr
integer(c_int), value :: device_num
```

Fortran

Constraints on Arguments

The value of *ptr* must be a valid host pointer or *NULL*. The *device_num* argument must be a conforming device number.

Binding

The binding task set for an **omp_target_is_present** region is the encountering task.

Effect

The **omp_target_is_present** routine returns *true* if *device_num* refers to the host device or if *ptr* refers to storage that has corresponding storage in the device data environment of device *device_num*. Otherwise, the routine returns *false*.

Restrictions

Restrictions to the **omp_target_is_present** routine are as follows.

- When called from within a **target** region the effect is unspecified.

Cross References

- **map** clause, see Section 5.8.2.
- **target** construct, see Section 13.8.

18.8.4 omp_target_is_accessible

Summary

The **omp_target_is_accessible** routine tests whether host memory is accessible from a given device.

Format

C / C++

```
int omp_target_is_accessible( const void *ptr, size_t size,
                              int device_num);
```

C / C++

Fortran

```
integer(c_int) function omp_target_is_accessible( &
               ptr, size, device_num) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t, c_int
type(c_ptr), value :: ptr
integer(c_size_t), value :: size
integer(c_int), value :: device_num
```

Fortran

Constraints on Arguments

The value of *ptr* must be a valid host pointer or *NULL*. The *device_num* argument must be a conforming device number.

Binding

The binding task set for an **omp_target_is_accessible** region is the encountering task.

Effect

This routine returns *true* if the storage of *size* bytes starting at the address given by *ptr* is accessible from device *device_num*. Otherwise, it returns *false*.

Restrictions

Restrictions to the **omp_target_is_accessible** routine are as follows.

- When called from within a **target** region the effect is unspecified.

Cross References

- **target** construct, see Section 13.8.

18.8.5 omp_target_memcpy

Summary

The **omp_target_memcpy** routine copies memory between any combination of host and device pointers.

Format

C / C++

```
int omp_target_memcpy(
  void *dst,
  const void *src,
  size_t length,
  size_t dst_offset,
  size_t src_offset,
  int dst_device_num,
  int src_device_num
);
```

C / C++

Fortran

```
integer(c_int) function omp_target_memcpy(dst, src, length, &
  dst_offset, src_offset, dst_device_num, src_device_num) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_int, c_size_t
type(c_ptr), value :: dst, src
integer(c_size_t), value :: length, dst_offset, src_offset
integer(c_int), value :: dst_device_num, src_device_num
```

Fortran

Constraints on Arguments

Each device pointer specified must be valid for the device on the same side of the copy. The *dst_device_num* and *src_device_num* arguments must be conforming device numbers.

Binding

The binding task set for an **omp_target_memcpy** region is the generating task, which is the *target task* generated by the call to the **omp_target_memcpy** routine.

Effect

This routine copies *length* bytes of memory at offset *src_offset* from *src* in the device data environment of device *src_device_num* to *dst* starting at offset *dst_offset* in the device data environment of device *dst_device_num*.

The **omp_target_memcpy** routine executes as if part of a target task that is generated by the call to the routine and that is an included task.

The return value is zero on success and non-zero on failure. This routine contains a task scheduling point.

Fortran

The **omp_target_memcpy** routine requires an explicit interface and so might not be provided in **omp_lib.h**.

Fortran

Execution Model Events

The *target-data-op-begin* event occurs before a thread initiates a data transfer in the **omp_target_memcpy** region.

The *target-data-op-end* event occurs after a thread initiates a data transfer in the **omp_target_memcpy** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_begin** as its endpoint argument for each occurrence of a *target-data-op-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_end** as its endpoint argument for each occurrence of a *target-data-op-end* event in that thread. These callbacks have type signature **ompt_callback_target_data_op_emi_t**.

A thread dispatches a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-op-end* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**.

Restrictions

Restrictions to the **omp_target_memcpy** routine are as follows.

- When called from within a **target** region the effect is unspecified.

Cross References

- **ompt_callback_target_data_op_t** or **ompt_callback_target_data_op_emi_t** callback type, see Section 19.5.2.25.
- **target** construct, see Section 13.8.

18.8.6 omp_target_memcpy_rect

Summary

The **omp_target_memcpy_rect** routine copies a rectangular subvolume from a multi-dimensional array to another multi-dimensional array. The **omp_target_memcpy_rect** routine performs a copy between any combination of host and device pointers.

Format

C / C++

```c
int omp_target_memcpy_rect(
  void *dst,
  const void *src,
  size_t element_size,
  int num_dims,
  const size_t *volume,
  const size_t *dst_offsets,
  const size_t *src_offsets,
  const size_t *dst_dimensions,
  const size_t *src_dimensions,
  int dst_device_num,
  int src_device_num
);
```

C / C++

Fortran

```fortran
integer(c_int) function omp_target_memcpy_rect(dst,src,element_size, &
  num_dims, volume, dst_offsets, src_offsets, dst_dimensions, src_dimensions, &
  dst_device_num, src_device_num) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_int, c_size_t
type(c_ptr), value :: dst, src
integer(c_size_t), value :: element_size
integer(c_int), value :: num_dims, dst_device_num, src_device_num
integer(c_size_t), intent(in) :: volume(*), dst_offsets(*), &
  src_offsets(*), dst_dimensions(*), src_dimensions(*)
```

Fortran

Constraints on Arguments

Each device pointer specified must be valid for the device on the same side of the copy. The *dst_device_num* and *src_device_num* arguments must be conforming device numbers.

The length of the offset and dimension arrays must be at least the value of *num_dims*. The value of *num_dims* must be between 1 and the implementation-defined limit, which must be at least three.

Fortran

Because the interface binds directly to a C language function the function assumes C memory ordering.

Fortran

Binding

The binding task set for an **omp_target_memcpy_rect** region is the generating task, which is the *target task* generated by the call to the **omp_target_memcpy_rect** routine.

Effect

This routine copies a rectangular subvolume of *src*, in the device data environment of device *src_device_num*, to *dst*, in the device data environment of device *dst_device_num*. The volume is specified in terms of the size of an element, number of dimensions, and constant arrays of length *num_dims*. The maximum number of dimensions supported is at least three; support for higher dimensionality is implementation defined. The *volume* array specifies the length, in number of elements, to copy in each dimension from *src* to *dst*. The *dst_offsets* (*src_offsets*) parameter specifies the number of elements from the origin of *dst* (*src*) in elements. The *dst_dimensions* (*src_dimensions*) parameter specifies the length of each dimension of *dst* (*src*).

The **omp_target_memcpy_rect** routine executes as if part of a target task that is generated by the call to the routine and that is an included task.

The routine returns zero if successful. Otherwise, it returns a non-zero value. The routine contains a task scheduling point.

An application can determine the inclusive number of dimensions supported by an implementation by passing *NULL* for both *dst* and *src*. The routine returns the number of dimensions supported by the implementation for the specified device numbers. No copy operation is performed.

Fortran

The **omp_target_memcpy_rect** routine requires an explicit interface and so might not be provided in **omp_lib.h**.

Fortran

Execution Model Events

The *target-data-op-begin* event occurs before a thread initiates a data transfer in the **omp_target_memcpy_rect** region.

The *target-data-op-end* event occurs after a thread initiates a data transfer in the **omp_target_memcpy_rect** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_begin** as its endpoint argument for each occurrence of a *target-data-op-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_end** as its endpoint argument for each occurrence of a *target-data-op-end* event in that thread. These callbacks have type signature **ompt_callback_target_data_op_emi_t**.

A thread dispatches a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-op-end* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**.

Restrictions

Restrictions to the **omp_target_memcpy_rect** routine are as follows.

- When called from within a **target** region the effect is unspecified.

Cross References

- **ompt_callback_target_data_op_t** or **ompt_callback_target_data_op_emi_t** callback type, see Section 19.5.2.25.
- **omp_get_num_devices** routine, see Section 18.7.4.
- **target** construct, see Section 13.8.

18.8.7 omp_target_memcpy_async

Summary

The **omp_target_memcpy_async** routine asynchronously performs a copy between any combination of host and device pointers.

Format

C / C++

```c
int omp_target_memcpy_async(
  void *dst,
  const void *src,
  size_t length,
  size_t dst_offset,
  size_t src_offset,
  int dst_device_num,
  int src_device_num,
  int depobj_count,
  omp_depend_t *depobj_list
);
```

C / C++

Fortran

```fortran
integer(c_int) function omp_target_memcpy_async(dst, src, length, &
  dst_offset, src_offset, dst_device_num, src_device_num, &
  depobj_count, depobj_list) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_int, c_size_t
type(c_ptr), value :: dst, src
integer(c_size_t), value :: length, dst_offset, src_offset
integer(c_int), value :: dst_device_num, src_device_num, depobj_count
integer(omp_depend_kind), optional :: depobj_list(*)
```

Fortran

Constraints on Arguments

Each device pointer specified must be valid for the device on the same side of the copy. The *dst_device_num* and *src_device_num* arguments must be conforming device numbers.

Binding

The binding task set for an **omp_target_memcpy_async** region is the generating task, which is the *target task* generated by the call to the **omp_target_memcpy_async** routine.

Effect

This routine performs an asynchronous memory copy where *length* bytes of memory at offset *src_offset* from *src* in the device data environment of device *src_device_num* are copied to *dst* starting at offset *dst_offset* in the device data environment of device *dst_device_num*.

The **omp_target_memcpy_async** routine executes as if part of a target task that is generated by the call to the routine and for which execution may be deferred.

Task dependences are expressed with zero or more **omp_depend_t** objects. The dependences are specified by passing the number of **omp_depend_t** objects followed by an array of **omp_depend_t** objects. The generated target task is not a dependent task if the program passes in a count of zero for *depobj_count*. *depojb_list* is ignored if the value of *depobj_count* is zero.

The routine returns zero if successful. Otherwise, it returns a non-zero value. The routine contains a task scheduling point.

Fortran

The **omp_target_memcpy_async** routine requires an explicit interface and so might not be provided in **omp_lib.h**.

Fortran

Execution Model Events

The *target-data-op-begin* event occurs before a thread initiates a data transfer in the **omp_target_memcpy_async** region.

The *target-data-op-end* event occurs after a thread initiates a data transfer in the **omp_target_memcpy_async** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_begin** as its endpoint argument for each occurrence of a *target-data-op-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_end** as its endpoint argument for each occurrence of a *target-data-op-end* event in that thread. These callbacks have type signature **ompt_callback_target_data_op_emi_t**.

A thread dispatches a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-op-end* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**.

Restrictions

Restrictions to the **omp_target_memcpy_async** routine are as follows.

- When called from within a **target** region the effect is unspecified.

Cross References

- **ompt_callback_target_data_op_t** or **ompt_callback_target_data_op_emi_t** callback type, see Section 19.5.2.25.
- Depend objects, see Section 15.9.2.
- **target** construct, see Section 13.8.

18.8.8 omp_target_memcpy_rect_async

Summary

The **omp_target_memcpy_rect_async** routine asynchronously performs a copy between any combination of host and device pointers.

Format

C / C++

```
int omp_target_memcpy_rect_async(
  void *dst,
  const void *src,
  size_t element_size,
  int num_dims,
  const size_t *volume,
  const size_t *dst_offsets,
  const size_t *src_offsets,
  const size_t *dst_dimensions,
  const size_t *src_dimensions,
  int dst_device_num,
  int src_device_num,
  int depobj_count,
  omp_depend_t *depobj_list
);
```

C / C++

Fortran

```
integer(c_int) function omp_target_memcpy_rect_async(dst, src, &
  element_size, num_dims, volume, dst_offsets, src_offsets, &
  dst_dimensions, src_dimensions, dst_device_num, src_device_num, &
  depobj_count, depobj_list) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_int, c_size_t
type(c_ptr), value :: dst, src
integer(c_size_t), value :: element_size
integer(c_int), value :: num_dims, dst_device_num, src_device_num, &
  depobj_count
integer(c_size_t), intent(in) :: volume(*), dst_offsets(*), &
  src_offsets(*), dst_dimensions(*), src_dimensions(*)
integer(omp_depobj_kind), optional :: depobj_list(*)
```

Fortran

Constraints on Arguments

Each device pointer specified must be valid for the device on the same side of the copy. The *dst_device_num* and *src_device_num* arguments must be conforming device numbers.

The length of the offset and dimension arrays must be at least the value of *num_dims*. The value of *num_dims* must be between 1 and the implementation-defined limit, which must be at least three.

Fortran

Because the interface binds directly to a C language function the function assumes C memory ordering.

Fortran

Binding

The binding task set for an **omp_target_memcpy_rect_async** region is the generating task, which is the *target task* generated by the call to the **omp_target_memcpy_rect_async** routine.

Effect

This routine copies a rectangular subvolume of *src*, in the device data environment of device *src_device_num*, to *dst*, in the device data environment of device *dst_device_num*. The volume is specified in terms of the size of an element, number of dimensions, and constant arrays of length *num_dims*. The maximum number of dimensions supported is at least three; support for higher dimensionality is implementation defined. The volume array specifies the length, in number of elements, to copy in each dimension from *src* to *dst*. The *dst_offsets* (*src_offsets*) parameter specifies the number of elements from the origin of *dst* (*src*) in elements. The *dst_dimensions* (*src_dimensions*) parameter specifies the length of each dimension of *dst* (*src*).

The **omp_target_memcpy_rect_async** routine executes as if part of a target task that is generated by the call to the routine and for which execution may be deferred.

Task dependences are expressed with zero or more **omp_depend_t** objects. The dependences are specified by passing the number of **omp_depend_t** objects followed by an array of **omp_depend_t** objects. The generated target task is not a dependent task if the program passes in a count of zero for *depobj_count*. *depobj_list* is ignored if the value of *depobj_count* is zero.

The routine returns zero if successful. Otherwise, it returns a non-zero value. The routine contains a task scheduling point.

An application can determine the number of inclusive dimensions supported by an implementation by passing *NULL* for both *dst* and *src*. The routine returns the number of dimensions supported by the implementation for the specified device numbers. No copy operation is performed.

Fortran

The **omp_target_memcpy_rect_async** routine requires an explicit interface and so might not be provided in **omp_lib.h**.

Fortran

Execution Model Events

The *target-data-op-begin* event occurs before a thread initiates a data transfer in the **omp_target_memcpy_rect_async** region.

The *target-data-op-end* event occurs after a thread initiates a data transfer in the **omp_target_memcpy_rect_async** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_begin** as its endpoint argument for each occurrence of a *target-data-op-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_end** as its endpoint argument for each occurrence of a *target-data-op-end* event in that thread. These callbacks have type signature **ompt_callback_target_data_op_emi_t**.

A thread dispatches a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-op-end* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**.

Restrictions

Restrictions to the **omp_target_memcpy_rect_async** routine are as follows.

- When called from within a **target** region the effect is unspecified.

Cross References

- **ompt_callback_target_data_op_t** or **ompt_callback_target_data_op_emi_t** callback type, see Section 19.5.2.25.
- Depend objects, see Section 15.9.2.
- **target** construct, see Section 13.8.

18.8.9 omp_target_associate_ptr

Summary

The **omp_target_associate_ptr** routine maps a device pointer, which may be returned from **omp_target_alloc** or implementation-defined runtime routines, to a host pointer.

Format

C / C++

```
int omp_target_associate_ptr(
  const void *host_ptr,
  const void *device_ptr,
  size_t size,
  size_t device_offset,
  int device_num
);
```

C / C++

Fortran

```
integer(c_int) function omp_target_associate_ptr(host_ptr, &
  device_ptr, size, device_offset, device_num) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t, c_int
type(c_ptr), value :: host_ptr, device_ptr
integer(c_size_t), value :: size, device_offset
integer(c_int), value :: device_num
```

Fortran

Constraints on Arguments

The value of *device_ptr* value must be a valid pointer to device memory for the device denoted by the value of *device_num*. The *device_num* argument must be a conforming device number.

Binding

The binding task set for an **omp_target_associate_ptr** region is the generating task, which is the *target task* generated by the call to the **omp_target_associate_ptr** routine.

Effect

The **omp_target_associate_ptr** routine associates a device pointer in the device data environment of device *device_num* with a host pointer such that when the host pointer appears in a subsequent **map** clause, the associated device pointer is used as the target for data motion associated with that host pointer. The *device_offset* parameter specifies the offset into *device_ptr* that is used as the base address for the device side of the mapping. The reference count of the resulting mapping will be infinite. After being successfully associated, the buffer to which the device pointer points is invalidated and accessing data directly through the device pointer results in unspecified behavior. The pointer can be retrieved for other uses by using the **omp_target_disassociate_ptr** routine to disassociate it .

The **omp_target_associate_ptr** routine executes as if part of a target task that is generated by the call to the routine and that is an included task.

The routine returns zero if successful. Otherwise it returns a non-zero value.

Only one device buffer can be associated with a given host pointer value and device number pair. Attempting to associate a second buffer will return non-zero. Associating the same pair of pointers on the same device with the same offset has no effect and returns zero. Associating pointers that share underlying storage will result in unspecified behavior. The **omp_target_is_present** function can be used to test whether a given host pointer has a corresponding variable in the device data environment.

Fortran

The **omp_target_associate_ptr** routine requires an explicit interface and so might not be provided in **omp_lib.h**.

Fortran

Execution Model Events

The *target-data-associate* event occurs before a thread initiates a device pointer association on a target device.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target_data_op** callback, or a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_beginend** as its endpoint argument for each occurrence of a *target-data-associate* event in that thread. These callbacks have type signature **ompt_callback_target_data_op_t** or **ompt_callback_target_data_op_emi_t**, respectively.

Restrictions

Restrictions to the **omp_target_associate_ptr** routine are as follows.

- When called from within a **target** region the effect is unspecified.

Cross References

- **ompt_callback_target_data_op_t** or **ompt_callback_target_data_op_emi_t** callback type, see Section 19.5.2.25.
- **map** clause, see Section 5.8.2.
- **omp_get_mapped_ptr** routine, see Section 18.8.11.
- **omp_target_alloc** routine, see Section 18.8.1.
- **omp_target_disassociate_ptr** routine, see Section 18.8.10.
- **omp_target_is_present** routine, see Section 18.8.3.
- **target** construct, see Section 13.8.

18.8.10 omp_target_disassociate_ptr

Summary

The **omp_target_disassociate_ptr** removes the associated pointer for a given device from a host pointer.

Format

C / C++

```
int omp_target_disassociate_ptr(const void *ptr, int device_num);
```

C / C++

Fortran

```
integer(c_int) function omp_target_disassociate_ptr(ptr, &
  device_num) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_int
type(c_ptr), value :: ptr
integer(c_int), value :: device_num
```

Fortran

Constraints on Arguments

The *device_num* argument must be a conforming device number.

Binding

The binding task set for an **omp_target_disassociate_ptr** region is the generating task, which is the *target task* generated by the call to the **omp_target_disassociate_ptr** routine.

Effect

The **omp_target_disassociate_ptr** removes the associated device data on device *device_num* from the presence table for host pointer *ptr*. A call to this routine on a pointer that is not *NULL* and does not have associated data on the given device results in unspecified behavior. The reference count of the mapping is reduced to zero, regardless of its current value.

The **omp_target_disassociate_ptr** routine executes as if part of a target task that is generated by the call to the routine and that is an included task.

The routine returns zero if successful. Otherwise it returns a non-zero value.

After a call to **omp_target_disassociate_ptr**, the contents of the device buffer are invalidated.

Fortran

The **omp_target_disassociate_ptr** routine requires an explicit interface and so might not be provided in **omp_lib.h**.

Fortran

Execution Model Events

The *target-data-disassociate* event occurs before a thread initiates a device pointer disassociation on a target device.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target_data_op** callback, or a registered **ompt_callback_target_data_op_emi** callback with **ompt_scope_beginend** as its endpoint argument for each occurrence of a *target-data-disassociate* event in that thread. These callbacks have type signature **ompt_callback_target_data_op_t** or **ompt_callback_target_data_op_emi_t**, respectively.

Restrictions

Restrictions to the **omp_target_disassociate_ptr** routine are as follows.

- When called from within a **target** region the effect is unspecified.

Cross References

- **ompt_callback_target_data_op_t** or **ompt_callback_target_data_op_emi_t** callback type, see Section 19.5.2.25.
- **omp_target_associate_ptr** routine, see Section 18.8.9.
- **target** construct, see Section 13.8.

18.8.11 omp_get_mapped_ptr

Summary

The **omp_get_mapped_ptr** routine returns the device pointer that is associated with a host pointer for a given device.

Format

C / C++

```
void * omp_get_mapped_ptr(const void *ptr, int device_num);
```

C / C++

Fortran

```
type(c_ptr) function omp_get_mapped_ptr(ptr, &
  device_num) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_int
type(c_ptr), value :: ptr
integer(c_int), value :: device_num
```

Fortran

Constraints on Arguments

The *device_num* argument must be a conforming device number.

Binding

The binding task set for an **omp_get_mapped_ptr** region is the encountering task.

Effect

The **omp_get_mapped_ptr** routine returns the associated device pointer on device *device_num*. A call to this routine for a pointer that is not *NULL* and does not have an associated pointer on the given device will return *NULL*.

The routine returns *NULL* if unsuccessful. Otherwise it returns the device pointer, which is *ptr* if *device_num* is the value returned by **omp_get_initial_device()**.

Fortran

The **omp_get_mapped_ptr** routine requires an explicit interface and so might not be provided in **omp_lib.h**.

Fortran

Execution Model Events

No events are associated with this routine.

Restrictions

Restrictions to the **omp_get_mapped_ptr** routine are as follows.

- When called from within a **target** region the effect is unspecified.

Cross References

- **omp_get_initial_device** routine, see Section 18.7.7.

18.9 Lock Routines

The OpenMP runtime library includes a set of general-purpose lock routines that can be used for synchronization. These general-purpose lock routines operate on OpenMP locks that are represented by OpenMP lock variables. OpenMP lock variables must be accessed only through the routines described in this section; programs that otherwise access OpenMP lock variables are non-conforming.

An OpenMP lock can be in one of the following states: *uninitialized*; *unlocked*; or *locked*. If a lock is in the *unlocked* state, a task can *set* the lock, which changes its state to *locked*. The task that sets the lock is then said to *own* the lock. A task that owns a lock can *unset* that lock, returning it to the *unlocked* state. A program in which a task unsets a lock that is owned by another task is non-conforming.

Two types of locks are supported: *simple locks* and *nestable locks*. A *nestable lock* can be set multiple times by the same task before being unset; a *simple lock* cannot be set if it is already owned by the task trying to set it. *Simple lock* variables are associated with *simple locks* and can only be passed to *simple lock* routines. *Nestable lock* variables are associated with *nestable locks* and can only be passed to *nestable lock* routines.

Each type of lock can also have a *synchronization hint* that contains information about the intended usage of the lock by the application code. The effect of the hint is implementation defined. An OpenMP implementation can use this hint to select a usage-specific lock, but hints do not change the mutual exclusion semantics of locks. A conforming implementation can safely ignore the hint.

Constraints on the state and ownership of the lock accessed by each of the lock routines are described with the routine. If these constraints are not met, the behavior of the routine is unspecified.

The OpenMP lock routines access a lock variable such that they always read and update the most current value of the lock variable. An OpenMP program does not need to include explicit **flush** directives to ensure that the lock variable's value is consistent among different tasks.

Binding

The binding thread set for all lock routine regions is all threads in the contention group. As a consequence, for each OpenMP lock, the lock routine effects relate to all tasks that call the routines, without regard to which teams in the contention group the threads that are executing the tasks belong.

Simple Lock Routines

C / C++

The type **omp_lock_t** represents a simple lock. For the following routines, a simple lock variable must be of **omp_lock_t** type. All simple lock routines require an argument that is a pointer to a variable of type **omp_lock_t**.

C / C++

Fortran

For the following routines, a simple lock variable must be an integer variable of **kind=omp_lock_kind**.

Fortran

The simple lock routines are as follows:

- The **omp_init_lock** routine initializes a simple lock;
- The **omp_init_lock_with_hint** routine initializes a simple lock and attaches a hint to it;
- The **omp_destroy_lock** routine uninitializes a simple lock;
- The **omp_set_lock** routine waits until a simple lock is available and then sets it;
- The **omp_unset_lock** routine unsets a simple lock; and

- The **omp_test_lock** routine tests a simple lock and sets it if it is available.

Nestable Lock Routines

C / C++

The type **omp_nest_lock_t** represents a nestable lock. For the following routines, a nestable lock variable must be of **omp_nest_lock_t** type. All nestable lock routines require an argument that is a pointer to a variable of type **omp_nest_lock_t**.

C / C++

Fortran

For the following routines, a nestable lock variable must be an integer variable of **kind=omp_nest_lock_kind**.

Fortran

The nestable lock routines are as follows:

- The **omp_init_nest_lock** routine initializes a nestable lock;
- The **omp_init_nest_lock_with_hint** routine initializes a nestable lock and attaches a hint to it;
- The **omp_destroy_nest_lock** routine uninitializes a nestable lock;
- The **omp_set_nest_lock** routine waits until a nestable lock is available and then sets it;
- The **omp_unset_nest_lock** routine unsets a nestable lock; and
- The **omp_test_nest_lock** routine tests a nestable lock and sets it if it is available.

Restrictions

Restrictions to OpenMP lock routines are as follows:

- The use of the same OpenMP lock in different contention groups results in unspecified behavior.

18.9.1 omp_init_lock and omp_init_nest_lock

Summary

These routines initialize an OpenMP lock without a hint.

Format

C / C++

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

C / C++

Fortran

```fortran
subroutine omp_init_lock(svar)
integer (kind=omp_lock_kind) svar

subroutine omp_init_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the uninitialized state through either routine is non-conforming.

Effect

The effect of these routines is to initialize the lock to the unlocked state; that is, no task owns the lock. In addition, the nesting count for a nestable lock is set to zero.

Execution Model Events

The *lock-init* event occurs in a thread that executes an **omp_init_lock** region after initialization of the lock, but before it finishes the region. The *nest-lock-init* event occurs in a thread that executes an **omp_init_nest_lock** region after initialization of the lock, but before it finishes the region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_lock_init** callback with **omp_sync_hint_none** as the *hint* argument and **ompt_mutex_lock** as the *kind* argument for each occurrence of a *lock-init* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_lock_init** callback with **omp_sync_hint_none** as the *hint* argument and **ompt_mutex_nest_lock** as the *kind* argument for each occurrence of a *nest-lock-init* event in that thread. These callbacks have the type signature **ompt_callback_mutex_acquire_t** and occur in the task that encounters the routine.

Cross References

- **ompt_callback_mutex_acquire_t**, see Section 19.5.2.14.

18.9.2 omp_init_lock_with_hint and omp_init_nest_lock_with_hint

Summary

These routines initialize an OpenMP lock with a hint. The effect of the hint is implementation-defined. The OpenMP implementation can ignore the hint without changing program semantics.

Format

C / C++

```
void omp_init_lock_with_hint(
  omp_lock_t *lock,
  omp_sync_hint_t hint
);
void omp_init_nest_lock_with_hint(
  omp_nest_lock_t *lock,
  omp_sync_hint_t hint
);
```

C / C++

Fortran

```
subroutine omp_init_lock_with_hint(svar, hint)
integer (kind=omp_lock_kind) svar
integer (kind=omp_sync_hint_kind) hint

subroutine omp_init_nest_lock_with_hint(nvar, hint)
integer (kind=omp_nest_lock_kind) nvar
integer (kind=omp_sync_hint_kind) hint
```

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the uninitialized state through either routine is non-conforming.

The second argument passed to these routines (*hint*) is a hint as described in Section 15.1.

Effect

The effect of these routines is to initialize the lock to the unlocked state and, optionally, to choose a specific lock implementation based on the hint. After initialization no task owns the lock. In addition, the nesting count for a nestable lock is set to zero.

Execution Model Events

The *lock-init-with-hint* event occurs in a thread that executes an **omp_init_lock_with_hint** region after initialization of the lock, but before it finishes the region. The *nest-lock-init-with-hint* event occurs in a thread that executes an **omp_init_nest_lock** region after initialization of the lock, but before it finishes the region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_lock_init** callback with the same value for its *hint* argument as the *hint* argument of the call to **omp_init_lock_with_hint** and **ompt_mutex_lock** as the *kind* argument for each occurrence of a *lock-init-with-hint* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_lock_init** callback with the same value for its *hint* argument as the *hint* argument of the call to **omp_init_nest_lock_with_hint** and **ompt_mutex_nest_lock** as the *kind* argument for each occurrence of a *nest-lock-init-with-hint* event in that thread. These callbacks have the type signature **ompt_callback_mutex_acquire_t** and occur in the task that encounters the routine.

Cross References

- **ompt_callback_mutex_acquire_t**, see Section 19.5.2.14.
- Synchronization Hints, see Section 15.1.

18.9.3 omp_destroy_lock and omp_destroy_nest_lock

Summary

These routines ensure that the OpenMP lock is uninitialized.

Format

C / C++

```
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

C / C++

Fortran

```
subroutine omp_destroy_lock(svar)
integer (kind=omp_lock_kind) svar

subroutine omp_destroy_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the unlocked state through either routine is non-conforming.

Effect

The effect of these routines is to change the state of the lock to uninitialized.

Execution Model Events

The *lock-destroy* event occurs in a thread that executes an **omp_destroy_lock** region before it finishes the region. The *nest-lock-destroy* event occurs in a thread that executes an **omp_destroy_nest_lock** region before it finishes the region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_lock_destroy** callback with **ompt_mutex_lock** as the *kind* argument for each occurrence of a *lock-destroy* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_lock_destroy** callback with **ompt_mutex_nest_lock** as the *kind* argument for each occurrence of a *nest-lock-destroy* event in that thread. These callbacks have the type signature **ompt_callback_mutex_t** and occur in the task that encounters the routine.

Cross References

- **ompt_callback_mutex_t**, see Section 19.5.2.15.

18.9.4 omp_set_lock and omp_set_nest_lock

Summary

These routines provide a means of setting an OpenMP lock. The calling task region behaves as if it was suspended until the lock can be set by this task.

Format

C / C++

```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

C / C++

Fortran

```
subroutine omp_set_lock(svar)
integer (kind=omp_lock_kind) svar

subroutine omp_set_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is in the uninitialized state through either routine is non-conforming. A simple lock accessed by **omp_set_lock** that is in the locked state must not be owned by the task that contains the call or deadlock will result.

Effect

Each of these routines has an effect equivalent to suspension of the task that is executing the routine until the specified lock is available.

Note – The semantics of these routines is specified *as if* they serialize execution of the region guarded by the lock. However, implementations may implement them in other ways provided that the isolation properties are respected so that the actual execution delivers a result that could arise from some serialization.

A simple lock is available if it is unlocked. Ownership of the lock is granted to the task that executes the routine.

A nestable lock is available if it is unlocked or if it is already owned by the task that executes the routine. The task that executes the routine is granted, or retains, ownership of the lock, and the nesting count for the lock is incremented.

Execution Model Events

The *lock-acquire* event occurs in a thread that executes an **omp_set_lock** region before the associated lock is requested. The *nest-lock-acquire* event occurs in a thread that executes an **omp_set_nest_lock** region before the associated lock is requested.

The *lock-acquired* event occurs in a thread that executes an **omp_set_lock** region after it acquires the associated lock but before it finishes the region. The *nest-lock-acquired* event occurs in a thread that executes an **omp_set_nest_lock** region if the thread did not already own the lock, after it acquires the associated lock but before it finishes the region.

The *nest-lock-owned* event occurs in a thread when it already owns the lock and executes an **omp_set_nest_lock** region. The event occurs after the nesting count is incremented but before the thread finishes the region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of a *lock-acquire* or *nest-lock-acquire* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of a *lock-acquired* or *nest-lock-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_nest_lock** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *nest-lock-owned* event in that thread. This callback has the type signature **ompt_callback_nest_lock_t**.

The above callbacks occur in the task that encounters the lock function. The *kind* argument of these callbacks is **ompt_mutex_lock** when the events arise from an **omp_set_lock** region while it is **ompt_mutex_nest_lock** when the events arise from an **omp_set_nest_lock** region.

Cross References

- **ompt_callback_mutex_acquire_t**, see Section 19.5.2.14.
- **ompt_callback_mutex_t**, see Section 19.5.2.15.
- **ompt_callback_nest_lock_t**, see Section 19.5.2.16.

18.9.5 omp_unset_lock and omp_unset_nest_lock

Summary

These routines provide the means of unsetting an OpenMP lock.

Format

C / C++

```
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

C / C++

Fortran

```
subroutine omp_unset_lock(svar)
integer (kind=omp_lock_kind) svar

subroutine omp_unset_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the locked state or that is not owned by the task that contains the call through either routine is non-conforming.

Effect

For a simple lock, the **omp_unset_lock** routine causes the lock to become unlocked.

For a nestable lock, the **omp_unset_nest_lock** routine decrements the nesting count, and causes the lock to become unlocked if the resulting nesting count is zero.

For either routine, if the lock becomes unlocked, and if one or more task regions were effectively suspended because the lock was unavailable, the effect is that one task is chosen and given ownership of the lock.

Execution Model Events

The *lock-release* event occurs in a thread that executes an **omp_unset_lock** region after it releases the associated lock but before it finishes the region. The *nest-lock-release* event occurs in a thread that executes an **omp_unset_nest_lock** region after it releases the associated lock but before it finishes the region.

The *nest-lock-held* event occurs in a thread that executes an **omp_unset_nest_lock** region before it finishes the region when the thread still owns the lock after the nesting count is decremented.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_released** callback with **ompt_mutex_lock** as the *kind* argument for each occurrence of a *lock-release* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_mutex_released** callback with **ompt_mutex_nest_lock** as the *kind* argument for each occurrence of a *nest-lock-release* event in that thread. These callbacks have the type signature **ompt_callback_mutex_t** and occur in the task that encounters the routine.

A thread dispatches a registered **ompt_callback_nest_lock** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *nest-lock-held* event in that thread. This callback has the type signature **ompt_callback_nest_lock_t**.

Cross References

- **ompt_callback_mutex_t**, see Section 19.5.2.15.
- **ompt_callback_nest_lock_t**, see Section 19.5.2.16.

18.9.6 omp_test_lock and omp_test_nest_lock

Summary

These routines attempt to set an OpenMP lock but do not suspend execution of the task that executes the routine.

Format

C / C++

```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

C / C++

Fortran

```
logical function omp_test_lock(svar)
integer (kind=omp_lock_kind) svar

integer function omp_test_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is in the uninitialized state through either routine is non-conforming. The behavior is unspecified if a simple lock accessed by **omp_test_lock** is in the locked state and is owned by the task that contains the call.

Effect

These routines attempt to set a lock in the same manner as **omp_set_lock** and **omp_set_nest_lock**, except that they do not suspend execution of the task that executes the routine.

For a simple lock, the **omp_test_lock** routine returns *true* if the lock is successfully set; otherwise, it returns *false*.

For a nestable lock, the **omp_test_nest_lock** routine returns the new nesting count if the lock is successfully set; otherwise, it returns zero.

Execution Model Events

The *lock-test* event occurs in a thread that executes an **omp_test_lock** region before the associated lock is tested. The *nest-lock-test* event occurs in a thread that executes an **omp_test_nest_lock** region before the associated lock is tested.

The *lock-test-acquired* event occurs in a thread that executes an **omp_test_lock** region before it finishes the region if the associated lock was acquired. The *nest-lock-test-acquired* event occurs in a thread that executes an **omp_test_nest_lock** region before it finishes the region if the associated lock was acquired and the thread did not already own the lock.

The *nest-lock-owned* event occurs in a thread that executes an **omp_test_nest_lock** region before it finishes the region after the nesting count is incremented if the thread already owned the lock.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of a *lock-test* or *nest-lock-test* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of a *lock-test-acquired* or *nest-lock-test-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_nest_lock** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *nest-lock-owned* event in that thread. This callback has the type signature **ompt_callback_nest_lock_t**.

The above callbacks occur in the task that encounters the lock function. The *kind* argument of these callbacks is **ompt_mutex_test_lock** when the events arise from an **omp_test_lock** region while it is **ompt_mutex_test_nest_lock** when the events arise from an **omp_test_nest_lock** region.

Cross References

- **ompt_callback_mutex_acquire_t**, see Section 19.5.2.14.
- **ompt_callback_mutex_t**, see Section 19.5.2.15.
- **ompt_callback_nest_lock_t**, see Section 19.5.2.16.

18.10 Timing Routines

This section describes routines that support a portable wall clock timer.

18.10.1 omp_get_wtime

Summary

The **omp_get_wtime** routine returns elapsed wall clock time in seconds.

Format

C / C++

```
double omp_get_wtime(void);
```

C / C++

Fortran

```
double precision function omp_get_wtime()
```

Fortran

Binding

The binding thread set for an **omp_get_wtime** region is the encountering thread. The routine's return value is not guaranteed to be consistent across any set of threads.

Effect

The **omp_get_wtime** routine returns a value equal to the elapsed wall clock time in seconds since some *time-in-the-past*. The actual *time-in-the-past* is arbitrary, but it is guaranteed not to change during the execution of the application program. The time returned is a *per-thread time*, so it is not required to be globally consistent across all threads that participate in an application.

18.10.2 omp_get_wtick

Summary

The **omp_get_wtick** routine returns the precision of the timer used by **omp_get_wtime**.

Format

C / C++

```
double omp_get_wtick(void);
```

C / C++

Fortran

```
double precision function omp_get_wtick()
```

Fortran

Binding

The binding thread set for an **omp_get_wtick** region is the encountering thread. The routine's return value is not guaranteed to be consistent across any set of threads.

Effect

The **omp_get_wtick** routine returns a value equal to the number of seconds between successive clock ticks of the timer used by **omp_get_wtime**.

18.11 Event Routine

This section describes a routine that supports OpenMP event objects.

Binding

The binding thread set for all event routine regions is the encountering thread.

18.11.1 omp_fulfill_event

Summary

This routine fulfills and destroys an OpenMP event.

Format

C / C++

```
void omp_fulfill_event(omp_event_handle_t event);
```

C / C++

Fortran

```
subroutine omp_fulfill_event(event)
integer (kind=omp_event_handle_kind) event
```

Fortran

Constraints on Arguments

A program that calls this routine on an event that was already fulfilled is non-conforming. A program that calls this routine with an event handle that was not created by the **detach** clause is non-conforming.

Effect

The effect of this routine is to fulfill the event associated with the event handle argument. The effect of fulfilling the event will depend on how the event was created. The event is destroyed and cannot be accessed after calling this routine, and the event handle becomes unassociated with any event.

Execution Model Events

The *task-fulfill* event occurs in a thread that executes an **omp_fulfill_event** region before the event is fulfilled if the OpenMP event object was created by a **detach** clause on a task.

Tool Callbacks

A thread dispatches a registered **ompt_callback_task_schedule** callback with *NULL* as its *next_task_data* argument while the argument *prior_task_data* binds to the detachable task for each occurrence of a *task-fulfill* event. If the *task-fulfill* event occurs before the detachable task finished the execution of the associated *structured-block*, the callback has **ompt_task_early_fulfill** as its *prior_task_status* argument; otherwise the callback has **ompt_task_late_fulfill** as its *prior_task_status* argument. This callback has type signature **ompt_callback_task_schedule_t**.

Restrictions

Restrictions to the **omp_fulfill_event** routine are as follows:

- The event handler passed to the routine must have been created by a thread in the same device as the thread that invoked the routine.

Cross References

- **ompt_callback_task_schedule_t**, see Section 19.5.2.10.
- **detach** clause, see Section 12.5.

C / C++

18.12 Interoperability Routines

The interoperability routines provide mechanisms to inspect the properties associated with an **omp_interop_t** object. Such objects may be initialized, destroyed or otherwise used by an **interop** construct. Additionally, an **omp_interop_t** object can be initialized to **omp_interop_none**, which is defined to be zero. An **omp_interop_t** object may only be accessed or modified through OpenMP directives and API routines.

An **omp_interop_t** object can be copied without affecting, or copying, the underlying state. Destruction of an **omp_interop_t** object destroys the state to which all copies of the object refer.

TABLE 18.1: Required Values of the **omp_interop_property_t** enum Type

enum name	contexts	name	property
omp_ipr_fr_id = -1	all	**fr_id**	An **intptr_t** value that represents the foreign runtime id of context
omp_ipr_fr_name = -2	all	**fr_name**	C string value that represents the foreign runtime name of context
omp_ipr_vendor = -3	all	**vendor**	An **intptr_t** that represents the vendor of context
omp_ipr_vendor_name = -4	all	**vendor_name**	C string value that represents the vendor of context
omp_ipr_device_num = -5	all	**device_num**	The OpenMP device ID for the device in the range 0 to **omp_get_num_devices()** inclusive
omp_ipr_platform = -6	*target*	**platform**	A foreign platform handle usually spanning multiple devices
omp_ipr_device = -7	*target*	**device**	A foreign device handle
omp_ipr_device_context = -8	*target*	**device_context**	A handle to an instance of a foreign device context
omp_ipr_targetsync = -9	*targetsync*	**targetsync**	A handle to a synchronization object of a foreign execution context
omp_ipr_first = -9			

OpenMP reserves all negative values for properties, as listed in Table 18.1; implementation-defined properties may use zero and positive values. The special property, **omp_ipr_first**, will always have the lowest property value which may change in future versions of this specification. Valid values and types for the properties that Table 18.1 lists are specified in the *OpenMP Additional Definitions* document or are implementation defined unless otherwise specified.

Table 18.2 lists the return codes used by routines that take an **int*** *ret_code* argument.

Binding

The binding task set for all interoperability routine regions is the generating task.

C / C++

18.12.1 omp_get_num_interop_properties

Summary

The **omp_get_num_interop_properties** routine retrieves the number of implementation-defined properties available for an **omp_interop_t** object.

TABLE 18.2: Required Values for the **omp_interop_rc_t** enum Type

enum name	description
omp_irc_no_value = 1	Parameters valid, no meaningful value available
omp_irc_success = 0	Successful, value is usable
omp_irc_empty = -1	The object provided is equal to **omp_interop_none**
omp_irc_out_of_range = -2	Property ID is out of range, see Table 18.1
omp_irc_type_int = -3	Property type is int; use **omp_get_interop_int**
omp_irc_type_ptr = -4	Property type is pointer; use **omp_get_interop_ptr**
omp_irc_type_str = -5	Property type is string; use **omp_get_interop_str**
omp_irc_other = -6	Other error; use **omp_get_interop_rc_desc**

Format

```
int omp_get_num_interop_properties(const omp_interop_t interop);
```

Effect

The **omp_get_num_interop_properties** routine returns the number of implementation-defined properties available for *interop*. The total number of properties available for *interop* is the returned value minus **omp_ipr_first**.

Cross References

- **interop** construct, see Section 14.1.

18.12.2 omp_get_interop_int

Summary

The **omp_get_interop_int** routine retrieves an integer property from an **omp_interop_t** object.

Format

```
omp_intptr_t omp_get_interop_int(const omp_interop_t interop,
                                 omp_interop_property_t property_id,
                                 int *ret_code);
```

Effect

The **omp_get_interop_int** routine returns the requested integer property, if available, and zero if an error occurs or no value is available.

If the *interop* is **omp_interop_none**, an empty error occurs.

If the *property_id* is smaller than **omp_ipr_first** or not smaller than **omp_get_num_interop_properties(*interop*)**, an out of range error occurs.

If the requested property value is not convertible into an integer value, a type error occurs.

If a non-null pointer is passed to *ret_code*, an **omp_interop_rc_t** value that indicates the return code is stored in the object to which *ret_code* points. If an error occurred, the stored value will be negative and it will match the error as defined in Table 18.2. On success, zero will be stored. If no error occurred but no meaningful value can be returned, **omp_irc_no_value**, which is one, will be stored.

Restrictions

Restrictions to the **omp_get_interop_int** routine are as follows:

- The behavior of the routine is unspecified if an invalid **omp_interop_t** object is provided.

Cross References

- **interop** construct, see Section 14.1.
- **omp_get_num_interop_properties** routine, see Section 18.12.1.

18.12.3 omp_get_interop_ptr

Summary

The **omp_get_interop_ptr** routine retrieves a pointer property from an **omp_interop_t** object.

Format

```
void* omp_get_interop_ptr(const omp_interop_t interop,
                          omp_interop_property_t property_id,
                          int *ret_code);
```

Effect

The **omp_get_interop_ptr** routine returns the requested pointer property, if available, and *NULL* if an error occurs or no value is available.

If the *interop* is **omp_interop_none**, an empty error occurs.

If the *property_id* is smaller than **omp_ipr_first** or not smaller than **omp_get_num_interop_properties(*interop*)**, an out of range error occurs.

If the requested property value is not convertible into a pointer value, a type error occurs.

If a non-null pointer is passed to *ret_code*, an **omp_interop_rc_t** value that indicates the return code is stored in the object to which the *ret_code* points. If an error occurred, the stored value will be negative and it will match the error as defined in Table 18.2. On success, zero will be stored. If no error occurred but no meaningful value can be returned, **omp_irc_no_value**, which is one, will be stored.

Restrictions

Restrictions to the **omp_get_interop_ptr** routine are as follows:

- The behavior of the routine is unspecified if an invalid **omp_interop_t** object is provided.
- Memory referenced by the pointer returned by the **omp_get_interop_ptr** routine is managed by the OpenMP implementation and should not be freed or modified.

Cross References

- **interop** construct, see Section 14.1.
- **omp_get_num_interop_properties** routine, see Section 18.12.1.

18.12.4 omp_get_interop_str

Summary

The **omp_get_interop_str** routine retrieves a string property from an **omp_interop_t** object.

Format

```
const char* omp_get_interop_str(const omp_interop_t interop,
                                omp_interop_property_t property_id,
                                int *ret_code);
```

Effect

The **omp_get_interop_str** routine returns the requested string property as a C string, if available, and *NULL* if an error occurs or no value is available.

If the *interop* is **omp_interop_none**, an empty error occurs.

If the *property_id* is smaller than **omp_ipr_first** or not smaller than **omp_get_num_interop_properties(*interop*)**, an out of range error occurs.

If the requested property value is not convertible into a string value, a type error occurs.

If a non-null pointer is passed to *ret_code*, an **omp_interop_rc_t** value that indicates the return code is stored in the object to which the *ret_code* points. If an error occurred, the stored value will be negative and it will match the error as defined in Table 18.2. On success, zero will be stored. If no error occurred but no meaningful value can be returned, **omp_irc_no_value**, which is one, will be stored.

Restrictions

Restrictions to the **omp_get_interop_str** routine are as follows:

- The behavior of the routine is unspecified if an invalid **omp_interop_t** object is provided.
- Memory referenced by the pointer returned by the **omp_get_interop_str** routine is managed by the OpenMP implementation and should not be freed or modified.

Cross References

- **interop** construct, see Section 14.1.
- **omp_get_num_interop_properties** routine, see Section 18.12.1.

18.12.5 omp_get_interop_name

Summary

The **omp_get_interop_name** routine retrieves a property name from an **omp_interop_t** object.

Format

```
const char* omp_get_interop_name(const omp_interop_t interop,
                                 omp_interop_property_t property_id)
  ;
```

Effect

The **omp_get_interop_name** routine returns the name of the property identified by *property_id* as a C string.

Property names for non-implementation defined properties are listed in Table 18.1.

If the *property_id* is smaller than **omp_ipr_first** or not smaller than **omp_get_num_interop_properties(*interop*)**, *NULL* is returned.

Restrictions

Restrictions to the **omp_get_interop_name** routine are as follows:

- The behavior of the routine is unspecified if an invalid object is provided.
- Memory referenced by the pointer returned by the **omp_get_interop_name** routine is managed by the OpenMP implementation and should not be freed or modified.

Cross References

- **interop** construct, see Section 14.1.
- **omp_get_num_interop_properties** routine, see Section 18.12.1.

18.12.6 omp_get_interop_type_desc

Summary

The **omp_get_interop_type_desc** routine retrieves a description of the type of a property associated with an **omp_interop_t** object.

Format

```
const char* omp_get_interop_type_desc(const omp_interop_t interop,
                                      omp_interop_property_t
    property_id);
```

Effect

The **omp_get_interop_type_desc** routine returns a C string that describes the type of the property identified by *property_id* in human-readable form. That may contain a valid C type declaration possibly followed by a description or name of the type.

If *interop* has the value **omp_interop_none**, *NULL* is returned.

If the *property_id* is smaller than **omp_ipr_first** or not smaller than **omp_get_num_interop_properties(*interop*)**, *NULL* is returned.

Restrictions

Restrictions to the **omp_get_interop_type_desc** routine are as follows:

- The behavior of the routine is unspecified if an invalid object is provided.
- Memory referenced by the pointer returned from the **omp_get_interop_type_desc** routine is managed by the OpenMP implementation and should not be freed or modified.

Cross References

- **interop** construct, see Section 14.1.
- **omp_get_num_interop_properties** routine, see Section 18.12.1.

18.12.7 omp_get_interop_rc_desc

Summary

The **omp_get_interop_rc_desc** routine retrieves a description of the return code associated with an **omp_interop_t** object.

Format

```
const char* omp_get_interop_rc_desc(const omp_interop_t interop,
                                    omp_interop_rc_t ret_code);
```

Effect

The **omp_get_interop_rc_desc** routine returns a C string that describes the return code *ret_code* in human-readable form.

Restrictions

Restrictions to the **omp_get_interop_rc_desc** routine are as follows:

- The behavior of the routine is unspecified if an invalid object is provided or if *ret_code* was not last written by an interoperability routine invoked with the **omp_interop_t** object *interop*.
- Memory referenced by the pointer returned by the **omp_get_interop_rc_desc** routine is managed by the OpenMP implementation and should not be freed or modified.

Cross References

- **interop** construct, see Section 14.1.
- **omp_get_num_interop_properties** routine, see Section 18.12.1.

18.13 Memory Management Routines

This section describes routines that support memory management on the current device.

Instances of memory management types must be accessed only through the routines described in this section; programs that otherwise access instances of these types are non-conforming.

18.13.1 Memory Management Types

The following type definitions are used by the memory management routines:

C / C++

```
typedef enum omp_alloctrait_key_t {
  omp_atk_sync_hint = 1,
  omp_atk_alignment = 2,
  omp_atk_access = 3,
  omp_atk_pool_size = 4,
  omp_atk_fallback = 5,
  omp_atk_fb_data = 6,
  omp_atk_pinned = 7,
  omp_atk_partition = 8
} omp_alloctrait_key_t;

typedef enum omp_alloctrait_value_t {
  omp_atv_false = 0,
  omp_atv_true = 1,
  omp_atv_contended = 3,
  omp_atv_uncontended = 4,
  omp_atv_serialized = 5,
  omp_atv_sequential = omp_atv_serialized, // (deprecated)
```

```c
  omp_atv_private = 6,
  omp_atv_all = 7,
  omp_atv_thread = 8,
  omp_atv_pteam = 9,
  omp_atv_cgroup = 10,
  omp_atv_default_mem_fb = 11,
  omp_atv_null_fb = 12,
  omp_atv_abort_fb = 13,
  omp_atv_allocator_fb = 14,
  omp_atv_environment = 15,
  omp_atv_nearest = 16,
  omp_atv_blocked = 17,
  omp_atv_interleaved = 18
} omp_alloctrait_value_t;

typedef struct omp_alloctrait_t {
  omp_alloctrait_key_t key;
  omp_uintptr_t value;
} omp_alloctrait_t;
```

C / C++

Fortran

```fortran

integer(kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_sync_hint = 1
integer(kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_alignment = 2
integer(kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_access = 3
integer(kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_pool_size = 4
integer(kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_fallback = 5
integer(kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_fb_data = 6
integer(kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_pinned = 7
integer(kind=omp_alloctrait_key_kind), &
   parameter :: omp_atk_partition = 8

integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_default = -1
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_false = 0
```

```
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_true = 1
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_contended = 3
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_uncontended = 4
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_serialized = 5
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_sequential = &
    omp_atv_serialized ! (deprecated)
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_private = 6
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_all = 7
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_thread = 8
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_pteam = 9
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_cgroup = 10
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_default_mem_fb = 11
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_null_fb = 12
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_abort_fb = 13
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_allocator_fb = 14
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_environment = 15
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_nearest = 16
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_blocked = 17
integer(kind=omp_alloctrait_val_kind), &
  parameter :: omp_atv_interleaved = 18

! omp_alloctrait might not be provided in omp_lib.h.
type omp_alloctrait
  integer(kind=omp_alloctrait_key_kind) key
  integer(kind=omp_alloctrait_val_kind) value
end type omp_alloctrait
```

```
integer(kind=omp_allocator_handle_kind), &
  parameter :: omp_null_allocator = 0
```

Fortran

18.13.2 omp_init_allocator

Summary

The **omp_init_allocator** routine initializes an allocator and associates it with a memory space.

Format

C / C++

```
omp_allocator_handle_t omp_init_allocator (
  omp_memspace_handle_t memspace,
  int ntraits,
  const omp_alloctrait_t traits[]
);
```

C / C++

Fortran

```
integer(kind=omp_allocator_handle_kind) &
function omp_init_allocator ( memspace, ntraits, traits )
integer(kind=omp_memspace_handle_kind),intent(in) :: memspace
integer,intent(in) :: ntraits
type(omp_alloctrait),intent(in) :: traits(*)
```

Fortran

Constraints on Arguments

The *memspace* argument must be one of the predefined memory spaces defined in Table 6.1.

If the *ntraits* argument is greater than zero then the *traits* argument must specify at least that many traits. If it specifies fewer than *ntraits* traits the behavior is unspecified.

Binding

The binding thread set for an **omp_init_allocator** region is all threads on a device. The effect of executing this routine is not related to any specific region that corresponds to any construct or API routine.

Effect

The **omp_init_allocator** routine creates a new allocator that is associated with the *memspace* memory space and returns a handle to it. All allocations through the created allocator will behave according to the allocator traits specified in the *traits* argument. The number of traits in the *traits* argument is specified by the *ntraits* argument. Specifying the same allocator trait more than once results in unspecified behavior. The routine returns a handle for the created allocator. If the special **omp_atv_default** value is used for a given trait, then its value will be the default value specified in Table 6.2 for that given trait.

If *memspace* is **omp_default_mem_space** and the **traits** argument is an empty set this routine will always return a handle to an allocator. Otherwise if an allocator based on the requirements cannot be created then the special **omp_null_allocator** handle is returned.

Restrictions

The restrictions to the **omp_init_allocator** routine are as follows:

- The use of an allocator returned by this routine on a device other than the one on which it was created results in unspecified behavior.
- Unless a **requires** directive with the **dynamic_allocators** clause is present in the same compilation unit, using this routine in a **target** region results in unspecified behavior.

Cross References

- Memory Allocators, see Section 6.2.
- Memory Spaces, see Section 6.1.

18.13.3 omp_destroy_allocator

Summary

The **omp_destroy_allocator** routine releases all resources used by the allocator handle.

Format

C / C++

```
void omp_destroy_allocator (omp_allocator_handle_t allocator);
```

C / C++

Fortran

```
subroutine omp_destroy_allocator ( allocator )
integer(kind=omp_allocator_handle_kind),intent(in) :: allocator
```

Fortran

Constraints on Arguments

The *allocator* argument must not represent a predefined memory allocator.

Binding

The binding thread set for an **omp_destroy_allocator** region is all threads on a device. The effect of executing this routine is not related to any specific region that corresponds to any construct or API routine.

Effect

The **omp_destroy_allocator** routine releases all resources used to implement the *allocator* handle.

If *allocator* is **omp_null_allocator** then this routine will have no effect.

Restrictions

The restrictions to the **omp_destroy_allocator** routine are as follows:

- Accessing any memory allocated by the *allocator* after this call results in unspecified behavior.
- Unless a **requires** directive with the **dynamic_allocators** clause is present in the same compilation unit, using this routine in a **target** region results in unspecified behavior.

Cross References

- Memory Allocators, see Section 6.2.

18.13.4 omp_set_default_allocator

Summary

The **omp_set_default_allocator** routine sets the default memory allocator to be used by allocation calls, **allocate** directives and **allocate** clauses that do not specify an allocator.

Format

C / C++

```
void omp_set_default_allocator (omp_allocator_handle_t allocator);
```

C / C++

Fortran

```
subroutine omp_set_default_allocator ( allocator )
integer(kind=omp_allocator_handle_kind),intent(in) :: allocator
```

Fortran

Constraints on Arguments

The *allocator* argument must be a valid memory allocator handle.

Binding

The binding task set for an **omp_set_default_allocator** region is the binding implicit task.

Effect

The effect of this routine is to set the value of the *def-allocator-var* ICV of the binding implicit task to the value specified in the *allocator* argument.

Cross References

- *def-allocator-var* ICV, see Section 2.
- Memory Allocators, see Section 6.2.
- **omp_alloc** routine, see Section 18.13.6.

18.13.5 omp_get_default_allocator

Summary

The **omp_get_default_allocator** routine returns a handle to the memory allocator to be used by allocation calls, **allocate** directives and **allocate** clauses that do not specify an allocator.

Format

C / C++

```
omp_allocator_handle_t omp_get_default_allocator (void);
```

C / C++

Fortran

```
integer(kind=omp_allocator_handle_kind)&
function omp_get_default_allocator ()
```

Fortran

Binding

The binding task set for an **omp_get_default_allocator** region is the binding implicit task.

Effect

The effect of this routine is to return the value of the *def-allocator-var* ICV of the binding implicit task.

Cross References

- *def-allocator-var* ICV, see Section 2.
- Memory Allocators, see Section 6.2.
- **omp_alloc** routine, see Section 18.13.6.

18.13.6 omp_alloc and omp_aligned_alloc

Summary

The **omp_alloc** and **omp_aligned_alloc** routines request a memory allocation from a memory allocator.

Format

C

```
void *omp_alloc(size_t size, omp_allocator_handle_t allocator);
void *omp_aligned_alloc(
  size_t alignment,
  size_t size,
  omp_allocator_handle_t allocator);
```

C

C++

```
void *omp_alloc(
  size_t size,
  omp_allocator_handle_t allocator=omp_null_allocator
);
void *omp_aligned_alloc(
  size_t alignment,
  size_t size,
  omp_allocator_handle_t allocator=omp_null_allocator
);
```

C++

Fortran

```
type(c_ptr) function omp_alloc(size, allocator) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t
integer(c_size_t), value :: size
integer(omp_allocator_handle_kind), value :: allocator

type(c_ptr) function omp_aligned_alloc(alignment, &
  size, allocator) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t
integer(c_size_t), value :: alignment, size
integer(omp_allocator_handle_kind), value :: allocator
```

Fortran

Constraints on Arguments

Unless **dynamic_allocators** appears on a **requires** directive in the same compilation unit, **omp_alloc** and **omp_aligned_alloc** invocations that appear in **target** regions must not pass **omp_null_allocator** as the *allocator* argument, which must be a constant expression that evaluates to one of the predefined memory allocator values.

The *alignment* argument to **omp_aligned_alloc** must be a power of two and the *size* argument must be a multiple of *alignment*.

Binding

The binding task set for an **omp_alloc** or **omp_aligned_alloc** region is the generating task.

Effect

The **omp_alloc** and **omp_aligned_alloc** routines request a memory allocation of *size* bytes from the specified memory allocator. If the *allocator* argument is **omp_null_allocator** the memory allocator used by the routines will be the one specified by the *def-allocator-var* ICV of the binding implicit task. Upon success they return a pointer to the allocated memory. Otherwise, the behavior that the **fallback** trait of the allocator specifies will be followed.

If *size* is 0, **omp_alloc** and **omp_aligned_alloc** will return *NULL*.

Memory allocated by **omp_alloc** will be byte-aligned to at least the maximum of the alignment required by **malloc** and the **alignment** trait of the allocator.

Memory allocated by **omp_aligned_alloc** will be byte-aligned to at least the maximum of the alignment required by **malloc**, the **alignment** trait of the allocator and the *alignment* argument value.

Fortran

The **omp_alloc** and **omp_aligned_alloc** routines require an explicit interface and so might not be provided in **omp_lib.h**.

Fortran

Cross References

- Memory allocators, see Section 6.2.

18.13.7 omp_free

Summary

The **omp_free** routine deallocates previously allocated memory.

Format

C

```c
void omp_free (void *ptr, omp_allocator_handle_t allocator);
```

C

C++

```cpp
void omp_free(
  void *ptr,
  omp_allocator_handle_t allocator=omp_null_allocator
);
```

C++

Fortran

```fortran
subroutine omp_free(ptr, allocator) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr
type(c_ptr), value :: ptr
integer(omp_allocator_handle_kind), value :: allocator
```

Fortran

Binding

The binding task set for an **omp_free** region is the generating task.

Effect

The **omp_free** routine deallocates the memory to which *ptr* points. The *ptr* argument must have been returned by an OpenMP allocation routine. If the *allocator* argument is specified it must be the memory allocator to which the allocation request was made. If the *allocator* argument is **omp_null_allocator** the implementation will determine that value automatically.

If *ptr* is *NULL*, no operation is performed.

Fortran

The **omp_free** routine requires an explicit interface and so might not be provided in **omp_lib.h**.

Fortran

Restrictions

The restrictions to the **omp_free** routine are as follows:

- Using **omp_free** on memory that was already deallocated or that was allocated by an allocator that has already been destroyed with **omp_destroy_allocator** results in unspecified behavior.

Cross References

- Memory allocators, see Section 6.2.

18.13.8 `omp_calloc` and `omp_aligned_calloc`

Summary

The **`omp_calloc`** and **`omp_aligned_calloc`** routines request a zero initialized memory allocation from a memory allocator.

Format

C

```
void *omp_calloc(
  size_t nmemb,
  size_t size,
  omp_allocator_handle_t allocator
);
void *omp_aligned_calloc(
  size_t alignment,
  size_t nmemb,
  size_t size,
  omp_allocator_handle_t allocator
);
```

C

C++

```
void *omp_calloc(
  size_t nmemb,
  size_t size,
  omp_allocator_handle_t allocator=omp_null_allocator
);
void *omp_aligned_calloc(
  size_t alignment,
  size_t nmemb,
  size_t size,
  omp_allocator_handle_t allocator=omp_null_allocator
);
```

C++

Fortran

```
type(c_ptr) function omp_calloc(nmemb, size, allocator) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t
integer(c_size_t), value :: nmemb, size
integer(omp_allocator_handle_kind), value :: allocator

type(c_ptr) function omp_aligned_calloc(alignment, nmemb, size, &
  allocator) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t
integer(c_size_t), value :: alignment, nmemb, size
integer(omp_allocator_handle_kind), value :: allocator
```

Fortran

Constraints on Arguments

Unless **dynamic_allocators** appears on a **requires** directive in the same compilation unit, **omp_calloc** and **omp_aligned_calloc** invocations that appear in **target** regions must not pass **omp_null_allocator** as the *allocator* argument, which must be a constant expression that evaluates to one of the predefined memory allocator values.

The *alignment* argument to **omp_aligned_calloc** must be a power of two and the *size* argument must be a multiple of *alignment*.

Binding

The binding task set for an **omp_calloc** or **omp_aligned_calloc** region is the generating task.

Effect

The **omp_calloc** and **omp_aligned_calloc** routines request a memory allocation from the specified memory allocator for an array of *nmemb* elements each of which has a size of *size* bytes. If the *allocator* argument is **omp_null_allocator** the memory allocator used by the routines will be the one specified by the *def-allocator-var* ICV of the binding implicit task. Upon success they return a pointer to the allocated memory. Otherwise, the behavior that the **fallback** trait of the allocator specifies will be followed. Any memory allocated by these routines will be set to zero before returning.

If either *nmemb* or *size* is 0, **omp_calloc** will return *NULL*.

Memory allocated by **omp_calloc** will be byte-aligned to at least the maximum of the alignment required by **malloc** and the **alignment** trait of the allocator.

Memory allocated by **omp_aligned_calloc** will be byte-aligned to at least the maximum of the alignment required by **malloc**, the **alignment** trait of the allocator and the *alignment* argument value.

Fortran

The **omp_calloc** and **omp_aligned_calloc** routines require an explicit interface and so might not be provided in **omp_lib.h**.

Fortran

Cross References

- Memory allocators, see Section 6.2.

18.13.9 omp_realloc

Summary

The **omp_realloc** routine deallocates previously allocated memory and requests a memory allocation from a memory allocator.

Format

C

```
void *omp_realloc(
  void *ptr,
  size_t size,
  omp_allocator_handle_t allocator,
  omp_allocator_handle_t free_allocator
);
```

C

C++

```
void *omp_realloc(
  void *ptr,
  size_t size,
  omp_allocator_handle_t allocator=omp_null_allocator,
  omp_allocator_handle_t free_allocator=omp_null_allocator
);
```

C++

Fortran

```
type(c_ptr) &
function omp_realloc(ptr, size, allocator, free_allocator) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t
type(c_ptr), value :: ptr
integer(c_size_t), value :: size
integer(omp_allocator_handle_kind), value :: allocator, free_allocator
```

Fortran

Constraints on Arguments

Unless a **dynamic_allocators** clause appears on a **requires** directive in the same compilation unit, **omp_realloc** invocations that appear in **target** regions must not pass **omp_null_allocator** as the *allocator* or *free_allocator* argument, which must be constant expressions that evaluate to one of the predefined memory allocator values.

Binding

The binding task set for an **omp_realloc** region is the generating task.

Effect

The **omp_realloc** routine deallocates the memory to which *ptr* points and requests a new memory allocation of *size* bytes from the specified memory allocator. If the *free_allocator* argument is specified, it must be the memory allocator to which the previous allocation request was made. If the *free_allocator* argument is **omp_null_allocator** the implementation will determine that value automatically. If the *allocator* argument is **omp_null_allocator** the behavior is as if the memory allocator that allocated the memory to which *ptr* argument points is passed to the *allocator* argument. Upon success it returns a (possibly moved) pointer to the allocated memory and the contents of the new object shall be the same as that of the old object prior to deallocation, up to the minimum size of old allocated size and *size*. Any bytes in the new object beyond the old allocated size will have unspecified values. If the allocation failed, the behavior that the **fallback** trait of the *allocator* specifies will be followed.

If *ptr* is *NULL*, **omp_realloc** will behave the same as **omp_alloc** with the same *size* and *allocator* arguments.

If *size* is 0, **omp_realloc** will return *NULL* and the old allocation will be deallocated.

If *size* is not 0, the old allocation will be deallocated if and only if the function returns a non-null value.

Memory allocated by **omp_realloc** will be byte-aligned to at least the maximum of the alignment required by **malloc** and the **alignment** trait of the allocator.

Fortran

The **omp_realloc** routine requires an explicit interface and so might not be provided in **omp_lib.h**.

Fortran

Restrictions

The restrictions to the **omp_realloc** routine are as follows:

- The *ptr* argument must have been returned by an OpenMP allocation routine.
- Using **omp_realloc** on memory that was already deallocated or that was allocated by an allocator that has already been destroyed with **omp_destroy_allocator** results in unspecified behavior.

Cross References

- Memory allocators, see Section 6.2.

18.14 Tool Control Routine

Summary

The **omp_control_tool** routine enables a program to pass commands to an active tool.

Format

C / C++

```
int omp_control_tool(int command, int modifier, void *arg);
```

C / C++

Fortran

```
integer function omp_control_tool(command, modifier)
integer (kind=omp_control_tool_kind) command
integer modifier
```

Fortran

Constraints on Arguments

The following enumeration type defines four standard commands. Table 18.3 describes the actions that these commands request from a tool.

C / C++

```
typedef enum omp_control_tool_t {
  omp_control_tool_start = 1,
  omp_control_tool_pause = 2,
  omp_control_tool_flush = 3,
  omp_control_tool_end = 4
} omp_control_tool_t;
```

C / C++

Fortran

```
integer (kind=omp_control_tool_kind), &
          parameter :: omp_control_tool_start = 1
integer (kind=omp_control_tool_kind), &
          parameter :: omp_control_tool_pause = 2
integer (kind=omp_control_tool_kind), &
          parameter :: omp_control_tool_flush = 3
integer (kind=omp_control_tool_kind), &
          parameter :: omp_control_tool_end = 4
```

Fortran

Tool-specific values for *command* must be greater or equal to 64. Tools must ignore *command* values that they are not explicitly designed to handle. Other values accepted by a tool for *command*, and any values for *modifier* and *arg* are tool-defined.

TABLE 18.3: Standard Tool Control Commands

Command	Action
omp_control_tool_start	Start or restart monitoring if it is off. If monitoring is already on, this command is idempotent. If monitoring has already been turned off permanently, this command will have no effect.
omp_control_tool_pause	Temporarily turn monitoring off. If monitoring is already off, it is idempotent.
omp_control_tool_flush	Flush any data buffered by a tool. This command may be applied whether monitoring is on or off.
omp_control_tool_end	Turn monitoring off permanently; the tool finalizes itself and flushes all output.

Binding

The binding task set for an **omp_control_tool** region is the generating task.

Effect

An OpenMP program may use **omp_control_tool** to pass commands to a tool. An application can use **omp_control_tool** to request that a tool starts or restarts data collection when a code region of interest is encountered, that a tool pauses data collection when leaving the region of interest, that a tool flushes any data that it has collected so far, or that a tool ends data collection. Additionally, **omp_control_tool** can be used to pass tool-specific commands to a particular tool.

The following types correspond to return values from **omp_control_tool**:

C / C++

```
typedef enum omp_control_tool_result_t {
  omp_control_tool_notool = -2,
  omp_control_tool_nocallback = -1,
  omp_control_tool_success = 0,
  omp_control_tool_ignored = 1
} omp_control_tool_result_t;
```

C / C++

Fortran

```
integer (kind=omp_control_tool_result_kind), &
        parameter :: omp_control_tool_notool = -2
integer (kind=omp_control_tool_result_kind), &
        parameter :: omp_control_tool_nocallback = -1
integer (kind=omp_control_tool_result_kind), &
        parameter :: omp_control_tool_success = 0
integer (kind=omp_control_tool_result_kind), &
        parameter :: omp_control_tool_ignored = 1
```

Fortran

If the OMPT interface state is inactive, the OpenMP implementation returns **omp_control_tool_notool**. If the OMPT interface state is active, but no callback is registered for the *tool-control* event, the OpenMP implementation returns **omp_control_tool_nocallback**. An OpenMP implementation may return other implementation-defined negative values strictly smaller than -64; an application may assume that any negative return value indicates that a tool has not received the command. A return value of **omp_control_tool_success** indicates that the tool has performed the specified command. A return value of **omp_control_tool_ignored** indicates that the tool has ignored the specified command. A tool may return other positive values strictly greater than 64 that are tool-defined.

Execution Model Events

The *tool-control* event occurs in the thread that encounters a call to **omp_control_tool** at a point inside its corresponding OpenMP region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_control_tool** callback for each occurrence of a *tool-control* event. The callback executes in the context of the call that occurs in the user program and has type signature **ompt_callback_control_tool_t**. The callback may return any non-negative value, which will be returned to the application by the OpenMP implementation as the return value of the **omp_control_tool** call that triggered the callback.

Arguments passed to the callback are those passed by the user to **omp_control_tool**. If the call is made in Fortran, the tool will be passed *NULL* as the third argument to the callback. If any of the four standard commands is presented to a tool, the tool will ignore the *modifier* and *arg* argument values.

Restrictions

Restrictions on access to the state of an OpenMP first-party tool are as follows:

- An application may access the tool state modified by an OMPT callback only by using **omp_control_tool**.

Cross References

- OMPT Interface, see Chapter 19
- **ompt_callback_control_tool_t**, see Section 19.5.2.29.

18.15 Environment Display Routine

Summary

The **omp_display_env** routine displays the OpenMP version number and the initial values of ICVs associated with the environment variables described in Chapter 21.

Format

C / C++

```
void omp_display_env(int verbose);
```

C / C++

Fortran

```
subroutine omp_display_env(verbose)
logical,intent(in) :: verbose
```

Fortran

Binding

The binding thread set for an **omp_display_env** region is the encountering thread.

Effect

Each time the **omp_display_env** routine is invoked, the runtime system prints the OpenMP version number and the initial values of the ICVs associated with the environment variables described in Chapter 21. The displayed values are the values of the ICVs after they have been modified according to the environment variable settings and before the execution of any OpenMP construct or API routine.

The display begins with `"OPENMP DISPLAY ENVIRONMENT BEGIN"`, followed by the **_OPENMP** version macro (or the **openmp_version** named constant for Fortran) and ICV values, in the format *NAME* '=' *VALUE*. *NAME* corresponds to the macro or environment variable name, optionally prepended with a bracketed *DEVICE*. *VALUE* corresponds to the value of the macro or ICV associated with this environment variable. Values are enclosed in single quotes. *DEVICE* corresponds to the device on which the value of the ICV is applied. The display is terminated with `"OPENMP DISPLAY ENVIRONMENT END"`.

For the **OMP_NESTED** environment variable, the printed value is *true* if the *max-active-levels-var* ICV is initialized to a value greater than 1; otherwise the printed value is *false*. The **OMP_NESTED** environment variable has been deprecated.

If the *verbose* argument is set to 0 (or *false* in Fortran), the runtime displays the OpenMP version number defined by the **_OPENMP** version macro (or the **openmp_version** named constant for Fortran) value and the initial ICV values for the environment variables listed in Chapter 21. If the *verbose* argument is set to 1 (or *true* for Fortran), the runtime may also display the values of vendor-specific ICVs that may be modified by vendor-specific environment variables.

Example output:

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP='201811'
  [host] OMP_SCHEDULE='GUIDED,4'
  [host] OMP_NUM_THREADS='4,3,2'
  [device] OMP_NUM_THREADS='2'
  [host,device] OMP_DYNAMIC='TRUE'
  [host] OMP_PLACES='{0:4},{4:4},{8:4},{12:4}'
  ...
OPENMP DISPLAY ENVIRONMENT END
```

Cross References

- **OMP_DISPLAY_ENV** environment variable, see Section 21.7.

19 OMPT Interface

This chapter describes OMPT, which is an interface for *first-party* tools. *First-party* tools are linked or loaded directly into the OpenMP program. OMPT defines mechanisms to initialize a tool, to examine OpenMP state associated with an OpenMP thread, to interpret the call stack of an OpenMP thread, to receive notification about OpenMP *events*, to trace activity on OpenMP target devices, to assess implementation-dependent details of an OpenMP implementation (such as supported states and mutual exclusion implementations), and to control a tool from an OpenMP application.

19.1 OMPT Interfaces Definitions

C / C++

A compliant implementation must supply a set of definitions for the OMPT runtime entry points, OMPT callback signatures, and the special data types of their parameters and return values. These definitions, which are listed throughout this chapter, and their associated declarations shall be provided in a header file named **omp-tools.h**. In addition, the set of definitions may specify other implementation-specific values.

The **ompt_start_tool** function is an external function with **C** linkage.

C / C++

19.2 Activating a First-Party Tool

To activate a tool, an OpenMP implementation first determines whether the tool should be initialized. If so, the OpenMP implementation invokes the initializer of the tool, which enables the tool to prepare to monitor execution on the host. The tool may then also arrange to monitor computation that executes on target devices. This section explains how the tool and an OpenMP implementation interact to accomplish these tasks.

19.2.1 ompt_start_tool

Summary

In order to use the OMPT interface provided by an OpenMP implementation, a tool must implement the **ompt_start_tool** function, through which the OpenMP implementation initializes the tool.

Format

C

```c
ompt_start_tool_result_t *ompt_start_tool(
  unsigned int omp_version,
  const char *runtime_version
);
```

C

Semantics

For a tool to use the OMPT interface that an OpenMP implementation provides, the tool must define a globally-visible implementation of the function **ompt_start_tool**. The tool indicates that it will use the OMPT interface that an OpenMP implementation provides by returning a non-null pointer to an **ompt_start_tool_result_t** structure from the **ompt_start_tool** implementation that it provides. The **ompt_start_tool_result_t** structure contains pointers to tool initialization and finalization callbacks as well as a tool data word that an OpenMP implementation must pass by reference to these callbacks. A tool may return *NULL* from **ompt_start_tool** to indicate that it will not use the OMPT interface in a particular execution.

A tool may use the *omp_version* argument to determine if it is compatible with the OMPT interface that the OpenMP implementation provides.

Description of Arguments

The argument *omp_version* is the value of the **_OPENMP** version macro associated with the OpenMP API implementation. This value identifies the OpenMP API version that an OpenMP implementation supports, which specifies the version of the OMPT interface that it supports.

The argument *runtime_version* is a version string that unambiguously identifies the OpenMP implementation.

Constraints on Arguments

The argument *runtime_version* must be an immutable string that is defined for the lifetime of a program execution.

Effect

If a tool returns a non-null pointer to an **ompt_start_tool_result_t** structure, an OpenMP implementation will call the tool initializer specified by the *initialize* field in this structure before beginning execution of any OpenMP construct or completing execution of any environment routine invocation; the OpenMP implementation will call the tool finalizer specified by the *finalize* field in this structure when the OpenMP implementation shuts down.

Cross References

- **ompt_start_tool_result_t**, see Section 19.4.1.

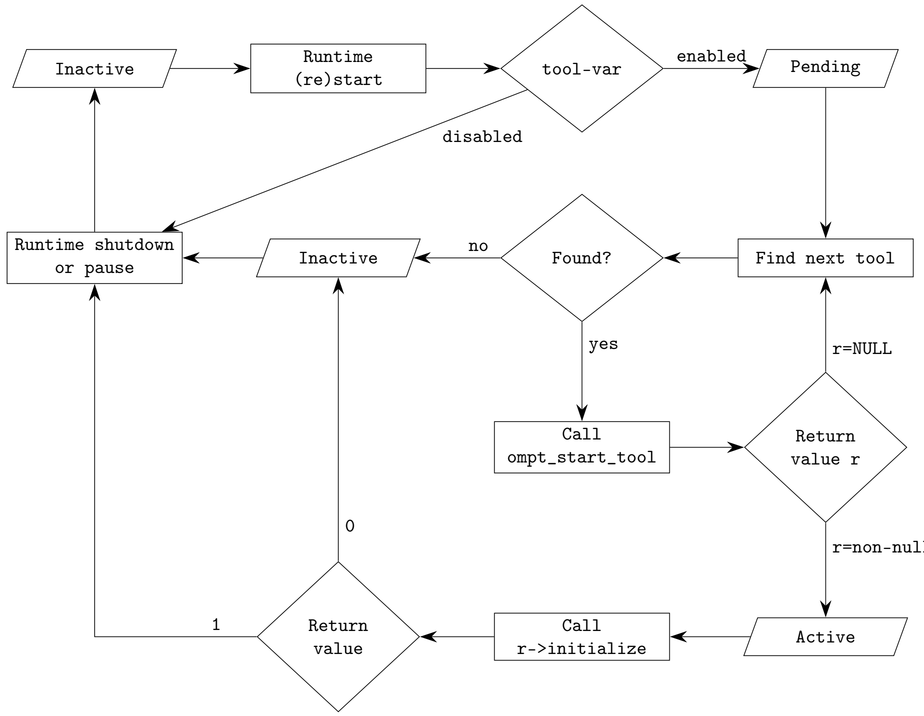

FIGURE 19.1: First-Party Tool Activation Flow Chart

19.2.2 Determining Whether a First-Party Tool Should be Initialized

An OpenMP implementation examines the *tool-var* ICV as one of its first initialization steps. If the value of *tool-var* is *disabled*, the initialization continues without a check for the presence of a tool and the functionality of the OMPT interface will be unavailable as the program executes. In this case, the OMPT interface state remains *inactive*.

Otherwise, the OMPT interface state changes to *pending* and the OpenMP implementation activates any first-party tool that it finds. A tool can provide a definition of **ompt_start_tool** to an OpenMP implementation in three ways:

- By statically-linking its definition of **ompt_start_tool** into an OpenMP application;
- By introducing a dynamically-linked library that includes its definition of **ompt_start_tool** into the application's address space; or
- By providing, in the *tool-libraries-var* ICV, the name of a dynamically-linked library that is appropriate for the architecture and operating system used by the application and that includes a definition of **ompt_start_tool**.

If the value of *tool-var* is *enabled*, the OpenMP implementation must check if a tool has provided an implementation of **ompt_start_tool**. The OpenMP implementation first checks if a tool-provided implementation of **ompt_start_tool** is available in the address space, either statically-linked into the application or in a dynamically-linked library loaded in the address space. If multiple implementations of **ompt_start_tool** are available, the OpenMP implementation will use the first tool-provided implementation of **ompt_start_tool** that it finds.

If the implementation does not find a tool-provided implementation of **ompt_start_tool** in the address space, it consults the *tool-libraries-var* ICV, which contains a (possibly empty) list of dynamically-linked libraries. As described in detail in Section 21.3.2, the libraries in *tool-libraries-var* are then searched for the first usable implementation of **ompt_start_tool** that one of the libraries in the list provides.

If the implementation finds a tool-provided definition of **ompt_start_tool**, it invokes that method; if a *NULL* pointer is returned, the OMPT interface state remains *pending* and the implementation continues to look for implementations of **ompt_start_tool**; otherwise a non-null pointer to an **ompt_start_tool_result_t** structure is returned, the OMPT interface state changes to *active* and the OpenMP implementation makes the OMPT interface available as the program executes. In this case, as the OpenMP implementation completes its initialization, it initializes the OMPT interface.

If no tool can be found, the OMPT interface state changes to *inactive*.

Cross References

- *tool-var* ICV, see Section 2.
- **ompt_start_tool** function, see Section 19.2.1.
- **ompt_start_tool_result_t** type, see Section 19.4.1.

19.2.3 Initializing a First-Party Tool

To initialize the OMPT interface, the OpenMP implementation invokes the tool initializer that is specified in the **ompt_start_tool_result_t** structure that is indicated by the non-null pointer that **ompt_start_tool** returns. The initializer is invoked prior to the occurrence of any OpenMP *event*.

A tool initializer, described in Section 19.5.1.1, uses the function specified in its *lookup* argument to look up pointers to OMPT interface runtime entry points that the OpenMP implementation provides; this process is described in Section 19.2.3.1. Typically, a tool initializer obtains a pointer to the **ompt_set_callback** runtime entry point with type signature **ompt_set_callback_t** and then uses this runtime entry point to register tool callbacks for OpenMP events, as described in Section 19.2.4.

A tool initializer may use the **ompt_enumerate_states** runtime entry point, which has type signature **ompt_enumerate_states_t**, to determine the thread states that an OpenMP implementation employs. Similarly, it may use the **ompt_enumerate_mutex_impls** runtime entry point, which has type signature **ompt_enumerate_mutex_impls_t**, to determine the mutual exclusion implementations that the OpenMP implementation employs.

If a tool initializer returns a non-zero value, the OMPT interface state remains *active* for the execution; otherwise, the OMPT interface state changes to *inactive*.

Cross References

- **ompt_callback_thread_begin_t** type, see Section 19.5.2.1.
- **ompt_enumerate_mutex_impls_t** type, see Section 19.6.1.2.
- **ompt_enumerate_states_t** type, see Section 19.6.1.1.
- **ompt_function_lookup_t** type, see Section 19.6.3.
- **ompt_initialize_t** type, see Section 19.5.1.1.
- **ompt_set_callback_t** type, see Section 19.6.1.3.
- **ompt_start_tool** function, see Section 19.2.1.
- **ompt_start_tool_result_t** type, see Section 19.4.1.

19.2.3.1 Binding Entry Points in the OMPT Callback Interface

Functions that an OpenMP implementation provides to support the OMPT interface are not defined as global function symbols. Instead, they are defined as runtime entry points that a tool can only identify through the *lookup* function that is provided as an argument with type signature **ompt_function_lookup_t** to the tool initializer. A tool can use this function to obtain a pointer to each of the runtime entry points that an OpenMP implementation provides to support the OMPT interface. Once a tool has obtained a *lookup* function, it may employ it at any point in the future.

For each runtime entry point in the OMPT interface for the host device, Table 19.1 provides the string name by which it is known and its associated type signature. Implementations can provide additional implementation-specific names and corresponding entry points. Any names that begin with **ompt_** are reserved names.

During initialization, a tool should look up each runtime entry point in the OMPT interface by name and bind a pointer maintained by the tool that can later be used to invoke the entry point. The entry points described in Table 19.1 enable a tool to assess the thread states and mutual exclusion implementations that an OpenMP implementation supports to register tool callbacks, to inspect registered callbacks, to introspect OpenMP state associated with threads, and to use tracing to monitor computations that execute on target devices.

Detailed information about each runtime entry point listed in Table 19.1 is included as part of the description of its type signature.

TABLE 19.1: OMPT Callback Interface Runtime Entry Point Names and Their Type Signatures

Entry Point String Name	Type signature
“ompt_enumerate_states”	ompt_enumerate_states_t
“ompt_enumerate_mutex_impls”	ompt_enumerate_mutex_impls_t
“ompt_set_callback”	ompt_set_callback_t
“ompt_get_callback”	ompt_get_callback_t
“ompt_get_thread_data”	ompt_get_thread_data_t
“ompt_get_num_places”	ompt_get_num_places_t
“ompt_get_place_proc_ids”	ompt_get_place_proc_ids_t
“ompt_get_place_num”	ompt_get_place_num_t
“ompt_get_partition_place_nums”	ompt_get_partition_place_nums_t
“ompt_get_proc_id”	ompt_get_proc_id_t
“ompt_get_state”	ompt_get_state_t
“ompt_get_parallel_info”	ompt_get_parallel_info_t
“ompt_get_task_info”	ompt_get_task_info_t
“ompt_get_task_memory”	ompt_get_task_memory_t
“ompt_get_num_devices”	ompt_get_num_devices_t
“ompt_get_num_procs”	ompt_get_num_procs_t
“ompt_get_target_info”	ompt_get_target_info_t
“ompt_get_unique_id”	ompt_get_unique_id_t
“ompt_finalize_tool”	ompt_finalize_tool_t

Cross References

- **ompt_enumerate_mutex_impls_t** type, see Section 19.6.1.2.
- **ompt_enumerate_states_t** type, see Section 19.6.1.1.
- **ompt_finalize_tool_t** type, see Section 19.6.1.19.
- **ompt_function_lookup_t** type, see Section 19.6.3.
- **ompt_get_callback_t** type, see Section 19.6.1.4.
- **ompt_get_num_devices_t** type, see Section 19.6.1.17.
- **ompt_get_num_places_t** type, see Section 19.6.1.7.
- **ompt_get_num_procs_t** type, see Section 19.6.1.6.
- **ompt_get_parallel_info_t** type, see Section 19.6.1.13.
- **ompt_get_partition_place_nums_t** type, see Section 19.6.1.10.
- **ompt_get_place_num_t** type, see Section 19.6.1.9.
- **ompt_get_place_proc_ids_t** type, see Section 19.6.1.8.
- **ompt_get_proc_id_t** type, see Section 19.6.1.11.
- **ompt_get_state_t** type, see Section 19.6.1.12.
- **ompt_get_target_info_t** type, see Section 19.6.1.16.
- **ompt_get_task_info_t** type, see Section 19.6.1.14.
- **ompt_get_task_memory_t** type, see Section 19.6.1.15.
- **ompt_get_thread_data_t** type, see Section 19.6.1.5.
- **ompt_get_unique_id_t** type, see Section 19.6.1.18.
- **ompt_set_callback_t** type, see Section 19.6.1.3.

19.2.4 Monitoring Activity on the Host with OMPT

To monitor the execution of an OpenMP program on the host device, a tool initializer must register to receive notification of events that occur as an OpenMP program executes. A tool can use the **ompt_set_callback** runtime entry point to register callbacks for OpenMP events. The return codes for **ompt_set_callback** use the **ompt_set_result_t** enumeration type. If the **ompt_set_callback** runtime entry point is called outside a tool initializer, registration of supported callbacks may fail with a return value of **ompt_set_error**.

All callbacks registered with **ompt_set_callback** or returned by **ompt_get_callback** use the dummy type signature **ompt_callback_t**.

TABLE 19.2: Callbacks for which **ompt_set_callback** Must Return **ompt_set_always**

Callback name
ompt_callback_thread_begin
ompt_callback_thread_end
ompt_callback_parallel_begin
ompt_callback_parallel_end
ompt_callback_task_create
ompt_callback_task_schedule
ompt_callback_implicit_task
ompt_callback_target
ompt_callback_target_emi
ompt_callback_target_data_op
ompt_callback_target_data_op_emi
ompt_callback_target_submit
ompt_callback_target_submit_emi
ompt_callback_control_tool
ompt_callback_device_initialize
ompt_callback_device_finalize
ompt_callback_device_load
ompt_callback_device_unload

For callbacks listed in Table 19.2, **ompt_set_always** is the only registration return code that is allowed. An OpenMP implementation must guarantee that the callback will be invoked every time that a runtime event that is associated with it occurs. Support for such callbacks is required in a minimal implementation of the OMPT interface.

For callbacks listed in Table 19.3, the **ompt_set_callback** runtime entry may return any non-error code. Whether an OpenMP implementation invokes a registered callback never, sometimes, or always is implementation defined. If registration for a callback allows a return code of **omp_set_never**, support for invoking such a callback may not be present in a minimal implementation of the OMPT interface. The return code from registering a callback indicates the implementation-defined level of support for the callback.

Two techniques reduce the size of the OMPT interface. First, in cases where events are naturally paired, for example, the beginning and end of a region, and the arguments needed by the callback at each endpoint are identical, a tool registers a single callback for the pair of events, with **ompt_scope_begin** or **ompt_scope_end** provided as an argument to identify for which endpoint the callback is invoked. Second, when a class of events is amenable to uniform treatment, OMPT provides a single callback for that class of events, for example, an **ompt_callback_sync_region_wait** callback is used for multiple kinds of synchronization regions, such as barrier, taskwait, and taskgroup regions. Some events, for example, **ompt_callback_sync_region_wait**, use both techniques.

TABLE 19.3: Callbacks for which **ompt_set_callback** May Return Any Non-Error Code

Callback name
ompt_callback_sync_region_wait
ompt_callback_mutex_released
ompt_callback_dependences
ompt_callback_task_dependence
ompt_callback_work
ompt_callback_master // (deprecated)
ompt_callback_masked
ompt_callback_target_map
ompt_callback_target_map_emi
ompt_callback_sync_region
ompt_callback_reduction
ompt_callback_lock_init
ompt_callback_lock_destroy
ompt_callback_mutex_acquire
ompt_callback_mutex_acquired
ompt_callback_nest_lock
ompt_callback_flush
ompt_callback_cancel
ompt_callback_dispatch

Cross References

- **ompt_get_callback_t** type, see Section 19.6.1.4.
- **ompt_set_callback_t** type, see Section 19.6.1.3.
- **ompt_set_result_t** type, see Section 19.4.4.2.

19.2.5 Tracing Activity on Target Devices with OMPT

A target device may or may not initialize a full OpenMP runtime system. Unless it does, monitoring activity on a device using a tool interface based on callbacks may not be possible. To accommodate such cases, the OMPT interface defines a monitoring interface for tracing activity on target devices. Tracing activity on a target device involves the following steps:

- To prepare to trace activity on a target device, a tool must register for an **ompt_callback_device_initialize** callback. A tool may also register for an **ompt_callback_device_load** callback to be notified when code is loaded onto a target device or an **ompt_callback_device_unload** callback to be notified when code is unloaded from a target device. A tool may also optionally register an **ompt_callback_device_finalize** callback.
- When an OpenMP implementation initializes a target device, the OpenMP implementation dispatches the device initialization callback of the tool on the host device. If the OpenMP implementation or target device does not support tracing, the OpenMP implementation passes *NULL* to the device initializer of the tool for its *lookup* argument; otherwise, the OpenMP implementation passes a pointer to a device-specific runtime entry point with type signature **ompt_function_lookup_t** to the device initializer of the tool.
- If a non-null *lookup* pointer is provided to the device initializer of the tool, the tool may use it to determine the runtime entry points in the tracing interface that are available for the device and may bind the returned function pointers to tool variables. Table 19.4 indicates the names of runtime entry points that may be available for a device; an implementations may provide additional implementation-defined names and corresponding entry points. The driver for the device provides the runtime entry points that enable a tool to control the trace collection interface of the device. The *native* trace format that the interface uses may be device specific and the available kinds of trace records are implementation defined. Some devices may allow a tool to collect traces of records in a standard format known as OMPT trace records. Each OMPT trace record serves as a substitute for an OMPT callback that cannot be made on the device. The fields in each trace record type are defined in the description of the callback that the record represents. If this type of record is provided then the *lookup* function returns values for the runtime entry points **ompt_set_trace_ompt** and **ompt_get_record_ompt**, which support collecting and decoding OMPT traces. If the native tracing format for a device is the OMPT format then tracing can be controlled using the runtime entry points for native or OMPT tracing.
- The tool uses the **ompt_set_trace_native** and/or the **ompt_set_trace_ompt** runtime entry point to specify what types of events or activities to monitor on the device. The

TABLE 19.4: OMPT Tracing Interface Runtime Entry Point Names and Their Type Signatures

Entry Point String Name	Type Signature
"**ompt_get_device_num_procs**"	**ompt_get_device_num_procs_t**
"**ompt_get_device_time**"	**ompt_get_device_time_t**
"**ompt_translate_time**"	**ompt_translate_time_t**
"**ompt_set_trace_ompt**"	**ompt_set_trace_ompt_t**
"**ompt_set_trace_native**"	**ompt_set_trace_native_t**
"**ompt_start_trace**"	**ompt_start_trace_t**
"**ompt_pause_trace**"	**ompt_pause_trace_t**
"**ompt_flush_trace**"	**ompt_flush_trace_t**
"**ompt_stop_trace**"	**ompt_stop_trace_t**
"**ompt_advance_buffer_cursor**"	**ompt_advance_buffer_cursor_t**
"**ompt_get_record_type**"	**ompt_get_record_type_t**
"**ompt_get_record_ompt**"	**ompt_get_record_ompt_t**
"**ompt_get_record_native**"	**ompt_get_record_native_t**
"**ompt_get_record_abstract**"	**ompt_get_record_abstract_t**

return codes for **ompt_set_trace_ompt** and **ompt_set_trace_native** use the **ompt_set_result_t** enumeration type. If the **ompt_set_trace_native** or the **ompt_set_trace_ompt** runtime entry point is called outside a device initializer, registration of supported callbacks may fail with a return code of **ompt_set_error**.

- The tool initiates tracing on the device by invoking **ompt_start_trace**. Arguments to **ompt_start_trace** include two tool callbacks through which the OpenMP implementation can manage traces associated with the device. One callback allocates a buffer in which the device can deposit trace events. The second callback processes a buffer of trace events from the device.

- If the device requires a trace buffer, the OpenMP implementation invokes the tool-supplied callback function on the host device to request a new buffer.

- The OpenMP implementation monitors the execution of OpenMP constructs on the device and records a trace of events or activities into a trace buffer. If possible, device trace records are marked with a *host_op_id*—an identifier that associates device activities with the target operation that the host initiated to cause these activities. To correlate activities on the host with activities on a device, a tool can register a **ompt_callback_target_submit_emi** callback. Before and after the host initiates creation of an initial task on a device associated with a structured block for a **target** construct, the OpenMP implementation dispatches the **ompt_callback_target_submit_emi** callback on the host in the thread that is executing the task that encounters the **target** construct. This callback provides the tool with a pair of identifiers: one that identifies the **target** region and a second that uniquely identifies the initial task associated with that region. These identifiers help the tool correlate activities on the target device with their **target** region.

- When appropriate, for example, when a trace buffer fills or needs to be flushed, the OpenMP implementation invokes the tool-supplied buffer completion callback to process a non-empty sequence of records in a trace buffer that is associated with the device.

- The tool-supplied buffer completion callback may return immediately, ignoring records in the trace buffer, or it may iterate through them using the **ompt_advance_buffer_cursor** entry point to inspect each record. A tool may use the **ompt_get_record_type** runtime entry point to inspect the type of the record at the current cursor position. Three runtime entry points (**ompt_get_record_ompt**, **ompt_get_record_native**, and **ompt_get_record_abstract**) allow tools to inspect the contents of some or all records in a trace buffer. The **ompt_get_record_native** runtime entry point uses the native trace format of the device. The **ompt_get_record_abstract** runtime entry point decodes the contents of a native trace record and summarizes them as an **ompt_record_abstract_t** record. The **ompt_get_record_ompt** runtime entry point can only be used to retrieve records in OMPT format.
- Once tracing has been started on a device, a tool may pause or resume tracing on the device at any time by invoking **ompt_pause_trace** with an appropriate flag value as an argument.
- A tool may invoke the **ompt_flush_trace** runtime entry point for a device at any time between device initialization and finalization to cause the device to flush pending trace records.
- At any time, a tool may use the **ompt_start_trace** runtime entry point to start tracing or the **ompt_stop_trace** runtime entry point to stop tracing on a device. When tracing is stopped on a device, the OpenMP implementation eventually gathers all trace records already collected on the device and presents them to the tool using the buffer completion callback.
- An OpenMP implementation can be shut down while device tracing is in progress.
- When an OpenMP implementation is shut down, it finalizes each device. Device finalization occurs in three steps. First, the OpenMP implementation halts any tracing in progress for the device. Second, the OpenMP implementation flushes all trace records collected for the device and uses the buffer completion callback associated with that device to present them to the tool. Finally, the OpenMP implementation dispatches any **ompt_callback_device_finalize** callback registered for the device.

Restrictions

Restrictions on tracing activity on devices are as follows:

- Implementation-defined names must not start with the prefix **ompt_**, which is reserved for the OpenMP specification.

Cross References

- **ompt_advance_buffer_cursor** runtime entry point, see Section 19.6.2.10.
- **ompt_callback_device_finalize_t** callback type, see Section 19.5.2.20.
- **ompt_callback_device_initialize_t** callback type, see Section 19.5.2.19.
- **ompt_flush_trace** runtime entry point, see Section 19.6.2.8.

- **ompt_get_device_num_procs** runtime entry point, see Section 19.6.2.1.
- **ompt_get_device_time** runtime entry point, see Section 19.6.2.2.
- **ompt_get_record_abstract** runtime entry point, see Section 19.6.2.14.
- **ompt_get_record_native** runtime entry point, see Section 19.6.2.13.
- **ompt_get_record_ompt** runtime entry point, see Section 19.6.2.12.
- **ompt_get_record_type** runtime entry point, see Section 19.6.2.11.
- **ompt_pause_trace** runtime entry point, see Section 19.6.2.7.
- **ompt_set_trace_native** runtime entry point, see Section 19.6.2.5.
- **ompt_set_trace_ompt** runtime entry point, see Section 19.6.2.4.
- **ompt_start_trace** runtime entry point, see Section 19.6.2.6.
- **ompt_stop_trace** runtime entry point, see Section 19.6.2.9.
- **ompt_translate_time** runtime entry point, see Section 19.6.2.3.

19.3 Finalizing a First-Party Tool

If the OMPT interface state is active, the tool finalizer, which has type signature **ompt_finalize_t** and is specified by the *finalize* field in the **ompt_start_tool_result_t** structure returned from the **ompt_start_tool** function, is called when the OpenMP implementation shuts down.

Cross References

- **ompt_finalize_t** callback type, see Section 19.5.1.2

19.4 OMPT Data Types

The C/C++ header file (**omp-tools.h**) provides the definitions of the types that are specified throughout this subsection.

19.4.1 Tool Initialization and Finalization

Summary

A tool's implementation of **ompt_start_tool** returns a pointer to an **ompt_start_tool_result_t** structure, which contains pointers to the tool's initialization and finalization callbacks as well as an **ompt_data_t** object for use by the tool.

Format

C / C++

```
typedef struct ompt_start_tool_result_t {
  ompt_initialize_t initialize;
  ompt_finalize_t finalize;
  ompt_data_t tool_data;
} ompt_start_tool_result_t;
```

C / C++

Restrictions

Restrictions to the **ompt_start_tool_result_t** type are as follows:

- The *initialize* and *finalize* callback pointer values in an **ompt_start_tool_result_t** structure that **ompt_start_tool** returns must be non-null.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_finalize_t** callback type, see Section 19.5.1.2.
- **ompt_initialize_t** callback type, see Section 19.5.1.1.
- **ompt_start_tool** function, see Section 19.2.1.

19.4.2 Callbacks

Summary

The **ompt_callbacks_t** enumeration type indicates the integer codes used to identify OpenMP callbacks when registering or querying them.

Format

C / C++

```
typedef enum ompt_callbacks_t {
  ompt_callback_thread_begin             = 1,
  ompt_callback_thread_end               = 2,
  ompt_callback_parallel_begin           = 3,
  ompt_callback_parallel_end             = 4,
  ompt_callback_task_create              = 5,
  ompt_callback_task_schedule            = 6,
  ompt_callback_implicit_task            = 7,
  ompt_callback_target                   = 8,
  ompt_callback_target_data_op           = 9,
  ompt_callback_target_submit            = 10,
  ompt_callback_control_tool             = 11,
  ompt_callback_device_initialize        = 12,
```

```
    ompt_callback_device_finalize            = 13,
    ompt_callback_device_load                = 14,
    ompt_callback_device_unload              = 15,
    ompt_callback_sync_region_wait           = 16,
    ompt_callback_mutex_released             = 17,
    ompt_callback_dependences                = 18,
    ompt_callback_task_dependence            = 19,
    ompt_callback_work                       = 20,
    ompt_callback_masked                     = 21,
    ompt_callback_master  /*(deprecated)*/   = ompt_callback_masked,
    ompt_callback_target_map                 = 22,
    ompt_callback_sync_region                = 23,
    ompt_callback_lock_init                  = 24,
    ompt_callback_lock_destroy               = 25,
    ompt_callback_mutex_acquire              = 26,
    ompt_callback_mutex_acquired             = 27,
    ompt_callback_nest_lock                  = 28,
    ompt_callback_flush                      = 29,
    ompt_callback_cancel                     = 30,
    ompt_callback_reduction                  = 31,
    ompt_callback_dispatch                   = 32,
    ompt_callback_target_emi                 = 33,
    ompt_callback_target_data_op_emi         = 34,
    ompt_callback_target_submit_emi          = 35,
    ompt_callback_target_map_emi             = 36,
    ompt_callback_error                      = 37
} ompt_callbacks_t;
```

C / C++

19.4.3 Tracing

OpenMP provides type definitions that support tracing with OMPT.

19.4.3.1 Record Type

Summary

The **ompt_record_t** enumeration type indicates the integer codes used to identify OpenMP trace record formats.

Format

C / C++

```
typedef enum ompt_record_t {
  ompt_record_ompt              = 1,
  ompt_record_native            = 2,
  ompt_record_invalid           = 3
} ompt_record_t;
```

C / C++

19.4.3.2 Native Record Kind

Summary

The **ompt_record_native_t** enumeration type indicates the integer codes used to identify OpenMP native trace record contents.

Format

C / C++

```
typedef enum ompt_record_native_t {
  ompt_record_native_info  = 1,
  ompt_record_native_event = 2
} ompt_record_native_t;
```

C / C++

19.4.3.3 Native Record Abstract Type

Summary

The **ompt_record_abstract_t** type provides an abstract trace record format that is used to summarize native device trace records.

Format

C / C++

```
typedef struct ompt_record_abstract_t {
  ompt_record_native_t rclass;
  const char *type;
  ompt_device_time_t start_time;
  ompt_device_time_t end_time;
  ompt_hwid_t hwid;
} ompt_record_abstract_t;
```

C / C++

Semantics

An **ompt_record_abstract_t** record contains information that a tool can use to process a native record that it may not fully understand. The *rclass* field indicates that the record is informational or that it represents an event; this information can help a tool determine how to present the record. The record *type* field points to a statically-allocated, immutable character string that provides a meaningful name that a tool can use to describe the event to a user. The *start_time* and *end_time* fields are used to place an event in time. The times are relative to the device clock. If an event does not have an associated *start_time* (*end_time*), the value of the *start_time* (*end_time*) field is **ompt_time_none**. The hardware identifier field, *hwid*, indicates the location on the device where the event occurred. A *hwid* may represent a hardware abstraction such as a core or a hardware thread identifier. The meaning of a *hwid* value for a device is implementation defined. If no hardware abstraction is associated with the record then the value of *hwid* is **ompt_hwid_none**.

19.4.3.4 Record Type

Summary

The **ompt_record_ompt_t** type provides a standard complete trace record format.

Format

C / C++

```
typedef struct ompt_record_ompt_t {
  ompt_callbacks_t type;
  ompt_device_time_t time;
  ompt_id_t thread_id;
  ompt_id_t target_id;
  union {
    ompt_record_thread_begin_t thread_begin;
    ompt_record_parallel_begin_t parallel_begin;
    ompt_record_parallel_end_t parallel_end;
    ompt_record_work_t work;
    ompt_record_dispatch_t dispatch;
    ompt_record_task_create_t task_create;
    ompt_record_dependences_t dependences;
    ompt_record_task_dependence_t task_dependence;
    ompt_record_task_schedule_t task_schedule;
    ompt_record_implicit_task_t implicit_task;
    ompt_record_masked_t masked;
    ompt_record_sync_region_t sync_region;
    ompt_record_mutex_acquire_t mutex_acquire;
    ompt_record_mutex_t mutex;
    ompt_record_nest_lock_t nest_lock;
    ompt_record_flush_t flush;
    ompt_record_cancel_t cancel;
```

```
    ompt_record_target_t target;
    ompt_record_target_data_op_t target_data_op;
    ompt_record_target_map_t target_map;
    ompt_record_target_kernel_t target_kernel;
    ompt_record_control_tool_t control_tool;
    ompt_record_error_t error;
  } record;
} ompt_record_ompt_t;
```

C / C++

Semantics

The field *type* specifies the type of record provided by this structure. According to the type, event specific information is stored in the matching *record* entry.

Restrictions

Restrictions to the **ompt_record_ompt_t** type are as follows:

- If *type* is set to **ompt_callback_thread_end_t** then the value of *record* is undefined.

19.4.4 Miscellaneous Type Definitions

This section describes miscellaneous types and enumerations used by the tool interface.

19.4.4.1 ompt_callback_t

Summary

Pointers to tool callback functions with different type signatures are passed to the **ompt_set_callback** runtime entry point and returned by the **ompt_get_callback** runtime entry point. For convenience, these runtime entry points expect all type signatures to be cast to a dummy type **ompt_callback_t**.

Format

C / C++

```
typedef void (*ompt_callback_t) (void);
```

C / C++

19.4.4.2 ompt_set_result_t

Summary

The **ompt_set_result_t** enumeration type corresponds to values that the **ompt_set_callback**, **ompt_set_trace_ompt** and **ompt_set_trace_native** runtime entry points return.

Format

C / C++

```
typedef enum ompt_set_result_t {
  ompt_set_error               = 0,
  ompt_set_never               = 1,
  ompt_set_impossible          = 2,
  ompt_set_sometimes           = 3,
  ompt_set_sometimes_paired = 4,
  ompt_set_always              = 5
} ompt_set_result_t;
```

C / C++

Semantics

Values of **ompt_set_result_t**, may indicate several possible outcomes. The **omp_set_error** value indicates that the associated call failed. Otherwise, the value indicates when an event may occur and, when appropriate, *dispatching* a callback event leads to the invocation of the callback. The **ompt_set_never** value indicates that the event will never occur or that the callback will never be invoked at runtime. The **ompt_set_impossible** value indicates that the event may occur but that tracing of it is not possible. The **ompt_set_sometimes** value indicates that the event may occur and, for an implementation-defined subset of associated event occurrences, will be traced or the callback will be invoked at runtime. The **ompt_set_sometimes_paired** value indicates the same result as **ompt_set_sometimes** and, in addition, that a callback with an *endpoint* value of **ompt_scope_begin** will be invoked if and only if the same callback with an *endpoint* value of **ompt_scope_end** will also be invoked sometime in the future. The **ompt_set_always** value indicates that, whenever an associated event occurs, it will be traced or the callback will be invoked.

Cross References

- Monitoring activity on the host with OMPT, see Section 19.2.4.
- **ompt_set_callback** runtime entry point, see Section 19.6.1.3.
- **ompt_set_trace_native** runtime entry point, see Section 19.6.2.5.
- **ompt_set_trace_ompt** runtime entry point, see Section 19.6.2.4.
- Tracing activity on target devices with OMPT, see Section 19.2.5.

19.4.4.3 ompt_id_t

Summary

The **ompt_id_t** type is used to provide various identifiers to tools.

Format

C / C++

```
typedef uint64_t ompt_id_t;
```

C / C++

Semantics

When tracing asynchronous activity on devices, identifiers enable tools to correlate target regions and operations that the host initiates with associated activities on a target device. In addition, OMPT provides identifiers to refer to parallel regions and tasks that execute on a device. These various identifiers are of type **ompt_id_t**.

ompt_id_none is defined as an instance of type **ompt_id_t** with the value 0.

Restrictions

Restrictions to the **ompt_id_t** type are as follows:

- Identifiers created on each device must be unique from the time an OpenMP implementation is initialized until it is shut down. Identifiers for each target region and target data operation instance that the host device initiates must be unique over time on the host. Identifiers for parallel and task region instances that execute on a device must be unique over time within that device.

19.4.4.4 ompt_data_t

Summary

The **ompt_data_t** type represents data associated with threads and with parallel and task regions.

Format

C / C++

```
typedef union ompt_data_t {
  uint64_t value;
  void *ptr;
} ompt_data_t;
```

C / C++

Semantics

The **ompt_data_t** type represents data that is reserved for tool use and that is related to a thread or to a parallel or task region. When an OpenMP implementation creates a thread or an instance of a parallel, **teams**, task, or target region, it initializes the associated **ompt_data_t** object with the value **ompt_data_none**, which is an instance of the type with the data and pointer fields equal to 0.

19.4.4.5 `ompt_device_t`

Summary

The **ompt_device_t** opaque object type represents a device.

Format

C / C++

```
typedef void ompt_device_t;
```

C / C++

19.4.4.6 `ompt_device_time_t`

Summary

The **ompt_device_time_t** type represents raw device time values.

Format

C / C++

```
typedef uint64_t ompt_device_time_t;
```

C / C++

Semantics

The **ompt_device_time_t** opaque object type represents raw device time values. **ompt_time_none** refers to an unknown or unspecified time and is defined as an instance of type **ompt_device_time_t** with the value 0.

19.4.4.7 `ompt_buffer_t`

Summary

The **ompt_buffer_t** opaque object type is a handle for a target buffer.

Format

C / C++

```
typedef void ompt_buffer_t;
```

C / C++

19.4.4.8 `ompt_buffer_cursor_t`

Summary

The **ompt_buffer_cursor_t** opaque type is a handle for a position in a target buffer.

Format

C / C++

```
typedef uint64_t ompt_buffer_cursor_t;
```

C / C++

19.4.4.9 ompt_dependence_t

Summary

The **ompt_dependence_t** type represents a task dependence.

Format

C / C++

```
typedef struct ompt_dependence_t {
  ompt_data_t variable;
  ompt_dependence_type_t dependence_type;
} ompt_dependence_t;
```

C / C++

Semantics

The **ompt_dependence_t** type is a structure that holds information about a depend clause. For task dependences, the *variable* field points to the storage location of the dependence. For *doacross* dependences, the *variable* field contains the value of a vector element that describes the dependence. The *dependence_type* field indicates the type of the dependence.

Cross References

- **ompt_dependence_type_t** type, see Section 19.4.4.24.

19.4.4.10 ompt_thread_t

Summary

The **ompt_thread_t** enumeration type defines the valid thread type values.

Format

C / C++

```
typedef enum ompt_thread_t {
  ompt_thread_initial                 = 1,
  ompt_thread_worker                  = 2,
  ompt_thread_other                   = 3,
  ompt_thread_unknown                 = 4
} ompt_thread_t;
```

C / C++

Semantics

Any *initial thread* has thread type **ompt_thread_initial**. All *OpenMP threads* that are not initial threads have thread type **ompt_thread_worker**. A thread that an OpenMP implementation uses but that does not execute user code has thread type **ompt_thread_other**. Any thread that is created outside an OpenMP implementation and that is not an *initial thread* has thread type **ompt_thread_unknown**.

19.4.4.11 ompt_scope_endpoint_t

Summary

The **ompt_scope_endpoint_t** enumeration type defines valid scope endpoint values.

Format

C / C++

```
typedef enum ompt_scope_endpoint_t {
  ompt_scope_begin                    = 1,
  ompt_scope_end                      = 2,
  ompt_scope_beginend                 = 3
} ompt_scope_endpoint_t;
```

C / C++

19.4.4.12 ompt_dispatch_t

Summary

The **ompt_dispatch_t** enumeration type defines the valid dispatch kind values.

Format

C / C++

```
typedef enum ompt_dispatch_t {
  ompt_dispatch_iteration             = 1,
  ompt_dispatch_section               = 2,
  ompt_dispatch_ws_loop_chunk         = 3,
  ompt_dispatch_taskloop_chunk        = 4,
  ompt_dispatch_distribute_chunk      = 5
} ompt_dispatch_t;
```

C / C++

19.4.4.13 ompt_dispatch_chunk_t

Summary

The **ompt_dispatch_chunk_t** type represents a the chunk information for a dispatched chunk.

Format

C / C++

```
typedef struct ompt_dispatch_chunk_t {
  uint64_t start;
  uint64_t iterations;
} ompt_dispatch_chunk_t;
```

C / C++

Semantics

The **ompt_dispatch_chunk_t** type is a structure that holds information about a chunk of logical iterations of a loop nest. The *start* field specifies the first logical iteration of the chunk and the *iterations* field specifies the number of iterations in the chunk. Whether the chunk of a taskloop is contiguous is implementation defined.

19.4.4.14 ompt_sync_region_t

Summary

The **ompt_sync_region_t** enumeration type defines the valid synchronization region kind values.

Format

C / C++

```
typedef enum ompt_sync_region_t {
  ompt_sync_region_barrier                       = 1, // deprecated
  ompt_sync_region_barrier_implicit              = 2, // deprecated
  ompt_sync_region_barrier_explicit              = 3,
  ompt_sync_region_barrier_implementation        = 4,
  ompt_sync_region_taskwait                      = 5,
  ompt_sync_region_taskgroup                     = 6,
  ompt_sync_region_reduction                     = 7,
  ompt_sync_region_barrier_implicit_workshare = 8,
  ompt_sync_region_barrier_implicit_parallel   = 9,
  ompt_sync_region_barrier_teams                 = 10
} ompt_sync_region_t;
```

C / C++

19.4.4.15 ompt_target_data_op_t

Summary

The **ompt_target_data_op_t** enumeration type defines the valid target data operation values.

Format

C / C++

```c
typedef enum ompt_target_data_op_t {
  ompt_target_data_alloc                      = 1,
  ompt_target_data_transfer_to_device         = 2,
  ompt_target_data_transfer_from_device       = 3,
  ompt_target_data_delete                     = 4,
  ompt_target_data_associate                  = 5,
  ompt_target_data_disassociate               = 6,
  ompt_target_data_alloc_async                = 17,
  ompt_target_data_transfer_to_device_async   = 18,
  ompt_target_data_transfer_from_device_async = 19,
  ompt_target_data_delete_async               = 20
} ompt_target_data_op_t;
```

C / C++

19.4.4.16 ompt_work_t

Summary

The **ompt_work_t** enumeration type defines the valid work type values.

Format

C / C++

```c
typedef enum ompt_work_t {
  ompt_work_loop                = 1,
  ompt_work_sections            = 2,
  ompt_work_single_executor     = 3,
  ompt_work_single_other        = 4,
  ompt_work_workshare           = 5,
  ompt_work_distribute          = 6,
  ompt_work_taskloop            = 7,
  ompt_work_scope               = 8,
  ompt_work_loop_static         = 10,
  ompt_work_loop_dynamic        = 11,
  ompt_work_loop_guided         = 12,
  ompt_work_loop_other          = 13
} ompt_work_t;
```

C / C++

19.4.4.17 ompt_mutex_t

Summary

The **ompt_mutex_t** enumeration type defines the valid mutex kind values.

Format

C / C++

```
typedef enum ompt_mutex_t {
  ompt_mutex_lock                       = 1,
  ompt_mutex_test_lock                  = 2,
  ompt_mutex_nest_lock                  = 3,
  ompt_mutex_test_nest_lock             = 4,
  ompt_mutex_critical                   = 5,
  ompt_mutex_atomic                     = 6,
  ompt_mutex_ordered                    = 7
} ompt_mutex_t;
```

C / C++

19.4.4.18 `ompt_native_mon_flag_t`

Summary

The **`ompt_native_mon_flag_t`** enumeration type defines the valid native monitoring flag values.

Format

C / C++

```
typedef enum ompt_native_mon_flag_t {
  ompt_native_data_motion_explicit    = 0x01,
  ompt_native_data_motion_implicit    = 0x02,
  ompt_native_kernel_invocation       = 0x04,
  ompt_native_kernel_execution        = 0x08,
  ompt_native_driver                  = 0x10,
  ompt_native_runtime                 = 0x20,
  ompt_native_overhead                = 0x40,
  ompt_native_idleness                = 0x80
} ompt_native_mon_flag_t;
```

C / C++

19.4.4.19 `ompt_task_flag_t`

Summary

The **`ompt_task_flag_t`** enumeration type defines valid task types.

Format

C / C++

```
typedef enum ompt_task_flag_t {
  ompt_task_initial                  = 0x00000001,
  ompt_task_implicit                 = 0x00000002,
  ompt_task_explicit                 = 0x00000004,
  ompt_task_target                   = 0x00000008,
  ompt_task_taskwait                 = 0x00000010,
  ompt_task_undeferred               = 0x08000000,
  ompt_task_untied                   = 0x10000000,
  ompt_task_final                    = 0x20000000,
  ompt_task_mergeable                = 0x40000000,
  ompt_task_merged                   = 0x80000000
} ompt_task_flag_t;
```

C / C++

Semantics

The **ompt_task_flag_t** enumeration type defines valid task type values. The least significant byte provides information about the general classification of the task. The other bits represent properties of the task.

19.4.4.20 ompt_task_status_t

Summary

The **ompt_task_status_t** enumeration type indicates the reason that a task was switched when it reached a task scheduling point.

Format

C / C++

```
typedef enum ompt_task_status_t {
  ompt_task_complete      = 1,
  ompt_task_yield         = 2,
  ompt_task_cancel        = 3,
  ompt_task_detach        = 4,
  ompt_task_early_fulfill = 5,
  ompt_task_late_fulfill  = 6,
  ompt_task_switch        = 7,
  ompt_taskwait_complete  = 8
} ompt_task_status_t;
```

C / C++

Semantics

The value **ompt_task_complete** of the **ompt_task_status_t** type indicates that the task that encountered the task scheduling point completed execution of the associated structured block and an associated *allow-completion* event was fulfilled. The value **ompt_task_yield** indicates that the task encountered a **taskyield** construct. The value **ompt_task_cancel** indicates that the task was canceled when it encountered an active cancellation point. The value **ompt_task_detach** indicates that a task for which the **detach** clause was specified completed execution of the associated structured block and is waiting for an *allow-completion* event to be fulfilled. The value **ompt_task_early_fulfill** indicates that the *allow-completion* event of the task was fulfilled before the task completed execution of the associated structured block. The value **ompt_task_late_fulfill** indicates that the *allow-completion* event of the task was fulfilled after the task completed execution of the associated structured block. The value **ompt_taskwait_complete** indicates completion of the dependent task that results from a **taskwait** construct with one or more **depend** clauses. The value **ompt_task_switch** is used for all other cases that a task was switched.

19.4.4.21 ompt_target_t

Summary

The **ompt_target_t** enumeration type defines the valid target type values.

Format

C / C++

```
typedef enum ompt_target_t {
  ompt_target                          = 1,
  ompt_target_enter_data               = 2,
  ompt_target_exit_data                = 3,
  ompt_target_update                   = 4,

  ompt_target_nowait                   = 9,
  ompt_target_enter_data_nowait        = 10,
  ompt_target_exit_data_nowait         = 11,
  ompt_target_update_nowait            = 12
} ompt_target_t;
```

C / C++

19.4.4.22 ompt_parallel_flag_t

Summary

The **ompt_parallel_flag_t** enumeration type defines valid invoker values.

Format

C / C++

```
typedef enum ompt_parallel_flag_t {
  ompt_parallel_invoker_program = 0x00000001,
  ompt_parallel_invoker_runtime = 0x00000002,
  ompt_parallel_league          = 0x40000000,
  ompt_parallel_team            = 0x80000000
} ompt_parallel_flag_t;
```

C / C++

Semantics

The **ompt_parallel_flag_t** enumeration type defines valid invoker values, which indicate how an outlined function is invoked.

The value **ompt_parallel_invoker_program** indicates that the outlined function associated with implicit tasks for the region is invoked directly by the application on the primary thread for a parallel region.

The value **ompt_parallel_invoker_runtime** indicates that the outlined function associated with implicit tasks for the region is invoked by the runtime on the primary thread for a parallel region.

The value **ompt_parallel_league** indicates that the callback is invoked due to the creation of a league of teams by a **teams** construct.

The value **ompt_parallel_team** indicates that the callback is invoked due to the creation of a team of threads by a **parallel** construct.

19.4.4.23 ompt_target_map_flag_t

Summary

The **ompt_target_map_flag_t** enumeration type defines the valid target map flag values.

Format

C / C++

```
typedef enum ompt_target_map_flag_t {
  ompt_target_map_flag_to             = 0x01,
  ompt_target_map_flag_from           = 0x02,
  ompt_target_map_flag_alloc          = 0x04,
  ompt_target_map_flag_release        = 0x08,
  ompt_target_map_flag_delete         = 0x10,
  ompt_target_map_flag_implicit       = 0x20,
  ompt_target_map_flag_always         = 0x40,
  ompt_target_map_flag_present        = 0x80,
  ompt_target_map_flag_close          = 0x100,
```

```
  ompt_target_map_flag_shared          = 0x200
} ompt_target_map_flag_t;
```

C / C++

Semantics

The **ompt_target_map_flag_** *map-type* flag is set if the mapping operations have that *map-type*. If the *map-type* for the mapping operations is **tofrom**, both the **ompt_target_map_flag_to** and **ompt_target_map_flag_from** flags are set. The **ompt_target_map_implicit** flag is set if the mapping operations result from implicit data-mapping rules. The **ompt_target_map_flag_** *map-type-modifier* flag is set if the mapping operations are specified with that *map-type-modifier*. The **ompt_target_map_flag_shared** flag is set if the original and corresponding storage are shared in the mapping operation.

19.4.4.24 ompt_dependence_type_t

Summary

The **ompt_dependence_type_t** enumeration type defines the valid task dependence type values.

Format

C / C++

```
typedef enum ompt_dependence_type_t {
  ompt_dependence_type_in               = 1,
  ompt_dependence_type_out              = 2,
  ompt_dependence_type_inout            = 3,
  ompt_dependence_type_mutexinoutset    = 4,
  ompt_dependence_type_source           = 5,
  ompt_dependence_type_sink             = 6,
  ompt_dependence_type_inoutset         = 7
} ompt_dependence_type_t;
```

C / C++

19.4.4.25 ompt_severity_t

Summary

The **ompt_severity_t** enumeration type defines the valid severity values.

Format

C / C++

```
typedef enum ompt_severity_t {
  ompt_warning              = 1,
  ompt_fatal                = 2
} ompt_severity_t;
```

C / C++

19.4.4.26 `ompt_cancel_flag_t`

Summary

The **ompt_cancel_flag_t** enumeration type defines the valid cancel flag values.

Format

C / C++

```
typedef enum ompt_cancel_flag_t {
  ompt_cancel_parallel       = 0x01,
  ompt_cancel_sections       = 0x02,
  ompt_cancel_loop           = 0x04,
  ompt_cancel_taskgroup      = 0x08,
  ompt_cancel_activated      = 0x10,
  ompt_cancel_detected       = 0x20,
  ompt_cancel_discarded_task = 0x40
} ompt_cancel_flag_t;
```

C / C++

19.4.4.27 `ompt_hwid_t`

Summary

The **ompt_hwid_t** opaque type is a handle for a hardware identifier for a target device.

Format

C / C++

```
typedef uint64_t ompt_hwid_t;
```

C / C++

Semantics

The **ompt_hwid_t** opaque type is a handle for a hardware identifier for a target device. **ompt_hwid_none** is an instance of the type that refers to an unknown or unspecified hardware identifier and that has the value 0. If no *hwid* is associated with an **ompt_record_abstract_t** then the value of *hwid* is **ompt_hwid_none**.

Cross References

- **ompt_record_abstract_t** type, see Section 19.4.3.3.

19.4.4.28 ompt_state_t

Summary

If the OMPT interface is in the *active* state then an OpenMP implementation must maintain *thread state* information for each thread. The thread state maintained is an approximation of the instantaneous state of a thread.

Format

C / C++

A thread state must be one of the values of the enumeration type **ompt_state_t** or an implementation-defined state value of 512 or higher.

```
typedef enum ompt_state_t {
  ompt_state_work_serial                      = 0x000,
  ompt_state_work_parallel                    = 0x001,
  ompt_state_work_reduction                   = 0x002,

  ompt_state_wait_barrier                     = 0x010, //
   deprecated
  ompt_state_wait_barrier_implicit_parallel   = 0x011,
  ompt_state_wait_barrier_implicit_workshare  = 0x012,
  ompt_state_wait_barrier_implicit            = 0x013, //
   deprecated
  ompt_state_wait_barrier_explicit            = 0x014,
  ompt_state_wait_barrier_implementation      = 0x015,
  ompt_state_wait_barrier_teams               = 0x016,

  ompt_state_wait_taskwait                    = 0x020,
  ompt_state_wait_taskgroup                   = 0x021,

  ompt_state_wait_mutex                       = 0x040,
  ompt_state_wait_lock                        = 0x041,
  ompt_state_wait_critical                    = 0x042,
  ompt_state_wait_atomic                      = 0x043,
  ompt_state_wait_ordered                     = 0x044,

  ompt_state_wait_target                      = 0x080,
  ompt_state_wait_target_map                  = 0x081,
  ompt_state_wait_target_update               = 0x082,

  ompt_state_idle                             = 0x100,
```

```
    ompt_state_overhead                              = 0x101,
    ompt_state_undefined                             = 0x102
} ompt_state_t;
```

C / C++

Semantics

A tool can query the OpenMP state of a thread at any time. If a tool queries the state of a thread that is not associated with OpenMP then the implementation reports the state as **ompt_state_undefined**.

The value **ompt_state_work_serial** indicates that the thread is executing code outside all **parallel** regions.

The value **ompt_state_work_parallel** indicates that the thread is executing code within the scope of a **parallel** region.

The value **ompt_state_work_reduction** indicates that the thread is combining partial reduction results from threads in its team. An OpenMP implementation may never report a thread in this state; a thread that is combining partial reduction results may have its state reported as **ompt_state_work_parallel** or **ompt_state_overhead**.

The value **ompt_state_wait_barrier_implicit_parallel** indicates that the thread is waiting at the implicit barrier at the end of a **parallel** region.

The value **ompt_state_wait_barrier_implicit_workshare** indicates that the thread is waiting at an implicit barrier at the end of a worksharing construct.

The value **ompt_state_wait_barrier_explicit** indicates that the thread is waiting in an explicit **barrier** region.

The value **ompt_state_wait_barrier_implementation** indicates that the thread is waiting in a barrier not required by the OpenMP standard but introduced by an OpenMP implementation.

The value **ompt_state_wait_barrier_teams** indicates that the thread is waiting at a barrier at the end of a **teams** region.

The value **ompt_state_wait_taskwait** indicates that the thread is waiting at a **taskwait** construct.

The value **ompt_state_wait_taskgroup** indicates that the thread is waiting at the end of a **taskgroup** construct.

The value **ompt_state_wait_mutex** indicates that the thread is waiting for a mutex of an unspecified type.

The value **ompt_state_wait_lock** indicates that the thread is waiting for a lock or nestable lock.

The value **ompt_state_wait_critical** indicates that the thread is waiting to enter a **critical** region.

The value **ompt_state_wait_atomic** indicates that the thread is waiting to enter an **atomic** region.

The value **ompt_state_wait_ordered** indicates that the thread is waiting to enter an **ordered** region.

The value **ompt_state_wait_target** indicates that the thread is waiting for a **target** region to complete.

The value **ompt_state_wait_target_map** indicates that the thread is waiting for a target data mapping operation to complete. An implementation may report **ompt_state_wait_target** for **target data** constructs.

The value **ompt_state_wait_target_update** indicates that the thread is waiting for a **target update** operation to complete. An implementation may report **ompt_state_wait_target** for **target update** constructs.

The value **ompt_state_idle** indicates that the thread is idle, that is, it is not part of an OpenMP team.

The value **ompt_state_overhead** indicates that the thread is in the overhead state at any point while executing within the OpenMP runtime, except while waiting at a synchronization point.

The value **ompt_state_undefined** indicates that the native thread is not created by the OpenMP implementation.

19.4.4.29 ompt_frame_t

Summary

The **ompt_frame_t** type describes procedure frame information for an OpenMP task.

Format

C / C++

```
typedef struct ompt_frame_t {
  ompt_data_t exit_frame;
  ompt_data_t enter_frame;
  int exit_frame_flags;
  int enter_frame_flags;
} ompt_frame_t;
```

C / C++

Semantics

Each **ompt_frame_t** object is associated with the task to which the procedure frames belong. Each non-merged initial, implicit, explicit, or target task with one or more frames on the stack of a native thread has an associated **ompt_frame_t** object.

The *exit_frame* field of an **ompt_frame_t** object contains information to identify the first procedure frame executing the task region. The *exit_frame* for the **ompt_frame_t** object associated with the *initial task* that is not nested inside any OpenMP construct is **ompt_data_none**.

The *enter_frame* field of an **ompt_frame_t** object contains information to identify the latest still active procedure frame executing the task region before entering the OpenMP runtime implementation or before executing a different task. If a task with frames on the stack is not executing implementation code in the OpenMP runtime, the value of *enter_frame* for the **ompt_frame_t** object associated with the task will be **ompt_data_none**.

For *exit_frame*, the *exit_frame_flags* and, for *enter_frame*, the *enter_frame_flags* field indicates that the provided frame information points to a runtime or an application frame address. The same fields also specify the kind of information that is provided to identify the frame, These fields are a disjunction of values in the **ompt_frame_flag_t** enumeration type.

The lifetime of an **ompt_frame_t** object begins when a task is created and ends when the task is destroyed. Tools should not assume that a frame structure remains at a constant location in memory throughout the lifetime of the task. A pointer to an **ompt_frame_t** object is passed to some callbacks; a pointer to the **ompt_frame_t** object of a task can also be retrieved by a tool at any time, including in a signal handler, by invoking the **ompt_get_task_info** runtime entry point (described in Section 19.6.1.14). A pointer to an **ompt_frame_t** object that a tool retrieved is valid as long as the tool does not pass back control to the OpenMP implementation.

Note – A monitoring tool that uses asynchronous sampling can observe values of *exit_frame* and *enter_frame* at inconvenient times. Tools must be prepared to handle **ompt_frame_t** objects observed just prior to when their field values will be set or cleared.

19.4.4.30 ompt_frame_flag_t

Summary

The **ompt_frame_flag_t** enumeration type defines valid frame information flags.

Format

C / C++

```
typedef enum ompt_frame_flag_t {
  ompt_frame_runtime          = 0x00,
  ompt_frame_application      = 0x01,
  ompt_frame_cfa              = 0x10,
  ompt_frame_framepointer     = 0x20,
  ompt_frame_stackaddress     = 0x30
} ompt_frame_flag_t;
```

C / C++

Semantics

The value **ompt_frame_runtime** of the **ompt_frame_flag_t** type indicates that a frame address is a procedure frame in the OpenMP runtime implementation. The value **ompt_frame_application** of the **ompt_frame_flag_t** type indicates that a frame address is a procedure frame in the OpenMP application.

Higher order bits indicate the kind of provided information that is unique for the particular frame pointer. The value **ompt_frame_cfa** indicates that a frame address specifies a *canonical frame address*. The value **ompt_frame_framepointer** indicates that a frame address provides the value of the frame pointer register. The value **ompt_frame_stackaddress** indicates that a frame address specifies a pointer address that is contained in the current stack frame.

19.4.4.31 ompt_wait_id_t

Summary

The **ompt_wait_id_t** type describes wait identifiers for an OpenMP thread.

Format

C / C++

```
typedef uint64_t ompt_wait_id_t;
```

C / C++

Semantics

Each thread maintains a *wait identifier* of type **ompt_wait_id_t**. When a task that a thread executes is waiting for mutual exclusion, the wait identifier of the thread indicates the reason that the thread is waiting. A wait identifier may represent a critical section *name*, a lock, a program variable accessed in an atomic region, or a synchronization object that is internal to an OpenMP implementation. When a thread is not in a wait state then the value of the wait identifier of the thread is undefined.

ompt_wait_id_none is defined as an instance of type **ompt_wait_id_t** with the value 0.

19.5 OMPT Tool Callback Signatures and Trace Records

The C/C++ header file (**omp-tools.h**) provides the definitions of the types that are specified throughout this subsection. Restrictions to the OpenMP tool callbacks are as follows:

Restrictions

- Tool callbacks may not use OpenMP directives or call any runtime library routines described in Section 18.
- Tool callbacks must exit by either returning to the caller or aborting.

19.5.1 Initialization and Finalization Callback Signature

19.5.1.1 ompt_initialize_t

Summary

A callback with type signature **ompt_initialize_t** initializes use of the OMPT interface.

Format

C / C++

```
typedef int (*ompt_initialize_t) (
  ompt_function_lookup_t lookup,
  int initial_device_num,
  ompt_data_t *tool_data
);
```

C / C++

Semantics

To use the OMPT interface, an implementation of **ompt_start_tool** must return a non-null pointer to an **ompt_start_tool_result_t** structure that contains a pointer to a tool initializer function with type signature **ompt_initialize_t**. An OpenMP implementation will call the initializer after fully initializing itself but before beginning execution of any OpenMP construct or runtime library routine.

The initializer returns a non-zero value if it succeeds; otherwise the OMPT interface state changes to *inactive* as described in Section 19.2.3.

Description of Arguments

The *lookup* argument is a callback to an OpenMP runtime routine that must be used to obtain a pointer to each runtime entry point in the OMPT interface. The *initial_device_num* argument provides the value of **omp_get_initial_device()**. The *tool_data* argument is a pointer to the *tool_data* field in the **ompt_start_tool_result_t** structure that **ompt_start_tool** returned.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_function_lookup_t** type, see Section 19.6.3.
- **ompt_start_tool** function, see Section 19.2.1.
- **ompt_start_tool_result_t** type, see Section 19.4.1.
- **omp_get_initial_device** routine, see Section 18.7.7.

19.5.1.2 ompt_finalize_t

Summary

A tool implements a finalizer with the type signature **ompt_finalize_t** to finalize its use of the OMPT interface.

Format

C / C++

```c
typedef void (*ompt_finalize_t) (
  ompt_data_t *tool_data
);
```

C / C++

Semantics

To use the OMPT interface, an implementation of **ompt_start_tool** must return a non-null pointer to an **ompt_start_tool_result_t** structure that contains a non-null pointer to a tool finalizer with type signature **ompt_finalize_t**. An OpenMP implementation must call the tool finalizer after the last OMPT *event* as the OpenMP implementation shuts down.

Description of Arguments

The *tool_data* argument is a pointer to the *tool_data* field in the **ompt_start_tool_result_t** structure returned by **ompt_start_tool**.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_start_tool** function, see Section 19.2.1.
- **ompt_start_tool_result_t** type, see Section 19.4.1.

19.5.2 Event Callback Signatures and Trace Records

This section describes the signatures of tool callback functions that an OMPT tool may register and that are called during runtime of an OpenMP program. An implementation may also provide a trace of events per device. Along with the callbacks, the following defines standard trace records. For the trace records, tool data arguments are replaced by an ID, which must be initialized by the OpenMP implementation. Each of *parallel_id*, *task_id*, and *thread_id* must be unique per target region. Tool implementations of callbacks are not required to be *async signal safe*.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_id_t** type, see Section 19.4.4.3.

19.5.2.1 ompt_callback_thread_begin_t

Summary

The **ompt_callback_thread_begin_t** type is used for callbacks that are dispatched when native threads are created.

Format

C / C++

```
typedef void (*ompt_callback_thread_begin_t) (
  ompt_thread_t thread_type,
  ompt_data_t *thread_data
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_thread_begin_t {
  ompt_thread_t thread_type;
} ompt_record_thread_begin_t;
```

C / C++

Description of Arguments

The *thread_type* argument indicates the type of the new thread: initial, worker, or other. The binding of the *thread_data* argument is the new thread.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_thread_t** type, see Section 19.4.4.10.
- **parallel** construct, see Section 10.1.
- **teams** construct, see Section 10.2.
- Initial task, see Section 12.8.

19.5.2.2 ompt_callback_thread_end_t

Summary

The **ompt_callback_thread_end_t** type is used for callbacks that are dispatched when native threads are destroyed.

Format

C / C++

```
typedef void (*ompt_callback_thread_end_t) (
  ompt_data_t *thread_data
);
```

C / C++

Description of Arguments

The binding of the *thread_data* argument is the thread that will be destroyed.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_record_ompt_t** type, see Section 19.4.3.4.
- **parallel** construct, see Section 10.1.
- **teams** construct, see Section 10.2.
- Initial task, see Section 12.8.

19.5.2.3 ompt_callback_parallel_begin_t

Summary

The **ompt_callback_parallel_begin_t** type is used for callbacks that are dispatched when a **parallel** or **teams** region starts.

Format

C / C++

```
typedef void (*ompt_callback_parallel_begin_t) (
  ompt_data_t *encountering_task_data,
  const ompt_frame_t *encountering_task_frame,
  ompt_data_t *parallel_data,
  unsigned int requested_parallelism,
  int flags,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_parallel_begin_t {
  ompt_id_t encountering_task_id;
  ompt_id_t parallel_id;
  unsigned int requested_parallelism;
  int flags;
  const void *codeptr_ra;
} ompt_record_parallel_begin_t;
```

C / C++

Description of Arguments

The binding of the *encountering_task_data* argument is the encountering task.

The *encountering_task_frame* argument points to the frame object that is associated with the encountering task. Accessing the frame object after the callback returned can cause a data race.

The binding of the *parallel_data* argument is the **parallel** or **teams** region that is beginning.

The *requested_parallelism* argument indicates the number of threads or teams that the user requested.

The *flags* argument indicates whether the code for the region is inlined into the application or invoked by the runtime and also whether the region is a **parallel** or **teams** region. Valid values for *flags* are a disjunction of elements in the enum **ompt_parallel_flag_t**.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_parallel_begin_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_frame_t** type, see Section 19.4.4.29.
- **ompt_parallel_flag_t** type, see Section 19.4.4.22.
- **parallel** construct, see Section 10.1.
- **teams** construct, see Section 10.2.

19.5.2.4 ompt_callback_parallel_end_t

Summary

The **ompt_callback_parallel_end_t** type is used for callbacks that are dispatched when a **parallel** or **teams** region ends.

Format

C / C++

```
typedef void (*ompt_callback_parallel_end_t) (
  ompt_data_t *parallel_data,
  ompt_data_t *encountering_task_data,
  int flags,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_parallel_end_t {
  ompt_id_t parallel_id;
  ompt_id_t encountering_task_id;
  int flags;
  const void *codeptr_ra;
} ompt_record_parallel_end_t;
```

C / C++

Description of Arguments

The binding of the *parallel_data* argument is the **parallel** or **teams** region that is ending.

The binding of the *encountering_task_data* argument is the encountering task.

The *flags* argument indicates whether the execution of the region is inlined into the application or invoked by the runtime and also whether it is a **parallel** or **teams** region. Values for *flags* are a disjunction of elements in the enum **ompt_parallel_flag_t**.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_parallel_end_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_parallel_flag_t** type, see Section 19.4.4.22.
- **parallel** construct, see Section 10.1.
- **teams** construct, see Section 10.2.

19.5.2.5 ompt_callback_work_t

Summary

The **ompt_callback_work_t** type is used for callbacks that are dispatched when worksharing regions and **taskloop** regions begin and end.

Format

C / C++

```
typedef void (*ompt_callback_work_t) (
  ompt_work_t work_type,
  ompt_scope_endpoint_t endpoint,
  ompt_data_t *parallel_data,
  ompt_data_t *task_data,
  uint64_t count,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_work_t {
  ompt_work_t work_type;
  ompt_scope_endpoint_t endpoint;
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  uint64_t count;
  const void *codeptr_ra;
} ompt_record_work_t;
```

C / C++

Description of Arguments

The *work_type* argument indicates the kind of region.

The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a scope.

The binding of the *parallel_data* argument is the current parallel region.

The binding of the *task_data* argument is the current task.

The *count* argument is a measure of the quantity of work involved in the construct. For a worksharing-loop or **taskloop** construct, *count* represents the number of iterations in the iteration space, which may be the result of collapsing several associated loops. For a **sections** construct, *count* represents the number of sections. For a **workshare** construct, *count* represents the units of work, as defined by the **workshare** construct. For a **single** or **scope** construct, *count* is always 1. When the *endpoint* argument signals the end of a scope, a *count* value of 0 indicates that the actual *count* value is not available.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_work_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Cross References

- Worksharing constructs, see Section 11.
- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_scope_endpoint_t** type, see Section 19.4.4.11.
- **ompt_work_t** type, see Section 19.4.4.16.
- **taskloop** construct, see Section 12.6.

19.5.2.6 `ompt_callback_dispatch_t`

Summary

The **ompt_callback_dispatch_t** type is used for callbacks that are dispatched when a thread begins to execute a section or loop iteration.

Format

C / C++

```
typedef void (*ompt_callback_dispatch_t) (
  ompt_data_t *parallel_data,
  ompt_data_t *task_data,
  ompt_dispatch_t kind,
  ompt_data_t instance
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_dispatch_t {
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  ompt_dispatch_t kind;
  ompt_data_t instance;
} ompt_record_dispatch_t;
```

C / C++

Description of Arguments

The binding of the *parallel_data* argument is the current parallel region.

The binding of the *task_data* argument is the implicit task that executes the structured block of the parallel region.

The *kind* argument indicates whether a loop iteration or a section is being dispatched.

If the *kind* argument is **ompt_dispatch_iteration**, the *value* field of the *instance* argument contains the logical iteration number. If the *kind* argument is **ompt_dispatch_section**, the *ptr* field of the *instance* argument contains a code address that identifies the structured block. In cases where a runtime routine implements the structured block associated with this callback, the *ptr* field of the *instance* argument contains the return address of the call to the runtime routine. In cases where the implementation of the structured block is inlined, the *ptr* field of the *instance* argument contains the return address of the invocation of this callback. If the *kind* argument is **ompt_dispatch_ws_loop_chunk**, **ompt_dispatch_taskloop_chunk** or **ompt_dispatch_distribute_chunk**, the *ptr* field of the *instance* argument points to a structure of type **ompt_dispatch_chunk_t** that contains the information for the chunk.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_dispatch_chunk_t** type, see Section 19.4.4.13.
- **ompt_dispatch_t** type, see Section 19.4.4.12.
- Worksharing-loop construct, see Section 11.5.
- **sections** and **section** constructs, see Section 11.3.
- **taskloop** construct, see Section 12.6.

19.5.2.7 ompt_callback_task_create_t

Summary

The **ompt_callback_task_create_t** type is used for callbacks that are dispatched when **task** regions are generated.

Format

C / C++

```
typedef void (*ompt_callback_task_create_t) (
  ompt_data_t *encountering_task_data,
  const ompt_frame_t *encountering_task_frame,
  ompt_data_t *new_task_data,
  int flags,
  int has_dependences,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_task_create_t {
  ompt_id_t encountering_task_id;
  ompt_id_t new_task_id;
  int flags;
  int has_dependences;
  const void *codeptr_ra;
} ompt_record_task_create_t;
```

C / C++

Description of Arguments

The binding of the *encountering_task_data* argument is the encountering task.

The *encountering_task_frame* argument points to the frame object associated with the encountering task. Accessing the frame object after the callback returned can cause a data race.

The binding of the *new_task_data* argument is the generated task.

The *flags* argument indicates the kind of task (explicit or target) that is generated. Values for *flags* are a disjunction of elements in the **ompt_task_flag_t** enumeration type.

The *has_dependences* argument is *true* if the generated task has dependences and *false* otherwise.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_task_create_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_frame_t** type, see Section 19.4.4.29.
- **ompt_task_flag_t** type, see Section 19.4.4.19.
- Initial task, see Section 12.8.
- **task** construct, see Section 12.5.

19.5.2.8 ompt_callback_dependences_t

Summary

The **ompt_callback_dependences_t** type is used for callbacks that are related to dependences and that are dispatched when new tasks are generated and when **ordered** constructs are encountered.

Format

C / C++

```
typedef void (*ompt_callback_dependences_t) (
  ompt_data_t *task_data,
  const ompt_dependence_t *deps,
  int ndeps
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_dependences_t {
  ompt_id_t task_id;
  ompt_dependence_t dep;
  int ndeps;
} ompt_record_dependences_t;
```

C / C++

Description of Arguments

The binding of the *task_data* argument is the generated task for a depend clause on a task construct, the target task for a depend clause on a target construct respectively depend object in an asynchronous runtime routine, or the encountering implicit task for a depend clause of the ordered construct.

The *deps* argument lists dependences of the new task or the dependence vector of the ordered construct. Dependences denoted with dependency objects are described in terms of their dependency semantics.

The *ndeps* argument specifies the length of the list passed by the *deps* argument. The memory for *deps* is owned by the caller; the tool cannot rely on the data after the callback returns.

The performance monitor interface for tracing activity on target devices provides one record per dependence.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_dependence_t** type, see Section 19.4.4.9.
- **depend** clause, see Section 15.9.5.
- **ordered** construct, see Section 15.9.7.

19.5.2.9 ompt_callback_task_dependence_t

Summary

The **ompt_callback_task_dependence_t** type is used for callbacks that are dispatched when unfulfilled task dependences are encountered.

Format

C / C++

```
typedef void (*ompt_callback_task_dependence_t) (
  ompt_data_t *src_task_data,
  ompt_data_t *sink_task_data
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_task_dependence_t {
  ompt_id_t src_task_id;
  ompt_id_t sink_task_id;
} ompt_record_task_dependence_t;
```

C / C++

Description of Arguments

The binding of the *src_task_data* argument is a running task with an outgoing dependence.

The binding of the *sink_task_data* argument is a task with an unsatisfied incoming dependence.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **depend** clause, see Section 15.9.5.

19.5.2.10 ompt_callback_task_schedule_t

Summary

The **ompt_callback_task_schedule_t** type is used for callbacks that are dispatched when task scheduling decisions are made.

Format

C / C++

```
typedef void (*ompt_callback_task_schedule_t) (
  ompt_data_t *prior_task_data,
  ompt_task_status_t prior_task_status,
  ompt_data_t *next_task_data
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_task_schedule_t {
  ompt_id_t prior_task_id;
  ompt_task_status_t prior_task_status;
  ompt_id_t next_task_id;
} ompt_record_task_schedule_t;
```

C / C++

Description of Arguments

The *prior_task_status* argument indicates the status of the task that arrived at a task scheduling point.

The binding of the *prior_task_data* argument is the task that arrived at the scheduling point.

The binding of the *next_task_data* argument is the task that is resumed at the scheduling point. This argument is *NULL* if the callback is dispatched for a *task-fulfill* event or if the callback signals completion of a taskwait construct.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_task_status_t** type, see Section 19.4.4.20.
- Task scheduling, see Section 12.9.

19.5.2.11 ompt_callback_implicit_task_t

Summary

The **ompt_callback_implicit_task_t** type is used for callbacks that are dispatched when initial tasks and implicit tasks are generated and completed.

Format

C / C++

```
typedef void (*ompt_callback_implicit_task_t) (
  ompt_scope_endpoint_t endpoint,
  ompt_data_t *parallel_data,
  ompt_data_t *task_data,
  unsigned int actual_parallelism,
  unsigned int index,
  int flags
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_implicit_task_t {
  ompt_scope_endpoint_t endpoint;
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  unsigned int actual_parallelism;
  unsigned int index;
  int flags;
} ompt_record_implicit_task_t;
```

C / C++

Description of Arguments

The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a scope.

The binding of the *parallel_data* argument is the current parallel or **teams** region. For the *implicit-task-end* and the *initial-task-end* events, this argument is *NULL*.

The binding of the *task_data* argument is the implicit task that executes the structured block of the parallel or **teams** region.

The *actual_parallelism* argument indicates the number of threads in the **parallel** region or the number of teams in the **teams** region. For initial tasks, that are not closely nested in a **teams** construct, this argument is **1**. For the *implicit-task-end* and the *initial-task-end* events, this argument is **0**.

The *index* argument indicates the thread number or team number of the calling thread, within the team or league that is executing the parallel or **teams** region to which the implicit task region binds. For initial tasks, that are not created by a **teams** construct, this argument is **1**.

The *flags* argument indicates the kind of task (initial or implicit).

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_scope_endpoint_t** enumeration type, see Section 19.4.4.11.
- **parallel** construct, see Section 10.1.
- **teams** construct, see Section 10.2.

19.5.2.12 `ompt_callback_masked_t`

Summary

The **ompt_callback_masked_t** type is used for callbacks that are dispatched when **masked** regions start and end.

Format

C / C++

```
typedef void (*ompt_callback_masked_t) (
  ompt_scope_endpoint_t endpoint,
  ompt_data_t *parallel_data,
  ompt_data_t *task_data,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_masked_t {
  ompt_scope_endpoint_t endpoint;
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  const void *codeptr_ra;
} ompt_record_masked_t;
```

C / C++

Description of Arguments

The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a scope.

The binding of the *parallel_data* argument is the current parallel region.

The binding of the *task_data* argument is the encountering task.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_masked_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Cross References

- **masked** construct, see Section 10.5.
- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_scope_endpoint_t** type, see Section 19.4.4.11.

19.5.2.13 ompt_callback_sync_region_t

Summary

The **ompt_callback_sync_region_t** type is used for callbacks that are dispatched when barrier regions, **taskwait** regions, and **taskgroup** regions begin and end and when waiting begins and ends for them as well as for when reductions are performed.

Format

C / C++

```
typedef void (*ompt_callback_sync_region_t) (
  ompt_sync_region_t kind,
  ompt_scope_endpoint_t endpoint,
  ompt_data_t *parallel_data,
  ompt_data_t *task_data,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_sync_region_t {
  ompt_sync_region_t kind;
  ompt_scope_endpoint_t endpoint;
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  const void *codeptr_ra;
} ompt_record_sync_region_t;
```

C / C++

Description of Arguments

The *kind* argument indicates the kind of synchronization.

The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a scope.

The binding of the *parallel_data* argument is the current parallel region. For the *implicit-barrier-end* event at the end of a parallel region this argument is *NULL*. For the *implicit-barrier-wait-begin* and *implicit-barrier-wait-end* event at the end of a parallel region, whether this argument is *NULL* or points to the parallel data of the current parallel region is implementation defined.

The binding of the *task_data* argument is the current task.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_sync_region_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Cross References

- **`ompt_data_t`** type, see Section 19.4.4.4.
- **`ompt_scope_endpoint_t`** type, see Section 19.4.4.11.
- **`ompt_sync_region_t`** type, see Section 19.4.4.14.
- **`barrier`** construct, see Section 15.3.1.
- Implicit barriers, see Section 15.3.2.
- **`taskgroup`** construct, see Section 15.4.
- **`taskwait`** construct, see Section 15.5.
- Properties common to all reduction clauses, see Section 5.5.6.

19.5.2.14 `ompt_callback_mutex_acquire_t`

Summary

The **`ompt_callback_mutex_acquire_t`** type is used for callbacks that are dispatched when locks are initialized, acquired and tested and when **`critical`** regions, **`atomic`** regions, and **`ordered`** regions are begun.

Format

C / C++

```
typedef void (*ompt_callback_mutex_acquire_t) (
  ompt_mutex_t kind,
  unsigned int hint,
  unsigned int impl,
  ompt_wait_id_t wait_id,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_mutex_acquire_t {
  ompt_mutex_t kind;
  unsigned int hint;
  unsigned int impl;
  ompt_wait_id_t wait_id;
  const void *codeptr_ra;
} ompt_record_mutex_acquire_t;
```

C / C++

Description of Arguments

The *kind* argument indicates the kind of mutual exclusion event.

The *hint* argument indicates the hint that was provided when initializing an implementation of mutual exclusion. If no hint is available when a thread initiates acquisition of mutual exclusion, the runtime may supply **omp_sync_hint_none** as the value for *hint*.

The *impl* argument indicates the mechanism chosen by the runtime to implement the mutual exclusion.

The *wait_id* argument indicates the object being awaited.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_mutex_acquire_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Cross References

- **ompt_mutex_t** type, see Section 19.4.4.17.
- **ompt_wait_id_t** type, see Section 19.4.4.31.
- **atomic** construct, see Section 15.8.4.
- **critical** construct, see Section 15.2.
- **omp_init_lock** and **omp_init_nest_lock** routines, see Section 18.9.1.
- **ordered** construct, see Section 15.9.7.

19.5.2.15 ompt_callback_mutex_t

Summary

The **ompt_callback_mutex_t** type is used for callbacks that indicate important synchronization events.

Format

C / C++

```
typedef void (*ompt_callback_mutex_t) (
  ompt_mutex_t kind,
  ompt_wait_id_t wait_id,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_mutex_t {
  ompt_mutex_t kind;
  ompt_wait_id_t wait_id;
  const void *codeptr_ra;
} ompt_record_mutex_t;
```

C / C++

Description of Arguments

The *kind* argument indicates the kind of mutual exclusion event.

The *wait_id* argument indicates the object being awaited.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_mutex_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Cross References

- **ompt_mutex_t** type, see Section 19.4.4.17.
- **ompt_wait_id_t** type, see Section 19.4.4.31.
- **atomic** construct, see Section 15.8.4.
- **critical** construct, see Section 15.2.
- **omp_destroy_lock** and **omp_destroy_nest_lock** routines, see Section 18.9.3.
- **omp_set_lock** and **omp_set_nest_lock** routines, see Section 18.9.4.
- **omp_test_lock** and **omp_test_nest_lock** routines, see Section 18.9.6.
- **omp_unset_lock** and **omp_unset_nest_lock** routines, see Section 18.9.5.
- **ordered** construct, see Section 15.9.7.

19.5.2.16 ompt_callback_nest_lock_t

Summary

The **ompt_callback_nest_lock_t** type is used for callbacks that indicate that a thread that owns a nested lock has performed an action related to the lock but has not relinquished ownership of it.

Format

C / C++

```
typedef void (*ompt_callback_nest_lock_t) (
  ompt_scope_endpoint_t endpoint,
  ompt_wait_id_t wait_id,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_nest_lock_t {
  ompt_scope_endpoint_t endpoint;
  ompt_wait_id_t wait_id;
  const void *codeptr_ra;
} ompt_record_nest_lock_t;
```

C / C++

Description of Arguments

The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a scope.

The *wait_id* argument indicates the object being awaited.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_nest_lock_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Cross References

- **ompt_scope_endpoint_t** type, see Section 19.4.4.11.
- **ompt_wait_id_t** type, see Section 19.4.4.31.
- **omp_set_nest_lock** routine, see Section 18.9.4.
- **omp_test_nest_lock** routine, see Section 18.9.6.
- **omp_unset_nest_lock** routine, see Section 18.9.5.

19.5.2.17 ompt_callback_flush_t

Summary

The **ompt_callback_flush_t** type is used for callbacks that are dispatched when **flush** constructs are encountered.

Format

C / C++

```
typedef void (*ompt_callback_flush_t) (
  ompt_data_t *thread_data,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_flush_t {
  const void *codeptr_ra;
} ompt_record_flush_t;
```

C / C++

Description of Arguments

The binding of the *thread_data* argument is the executing thread.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_flush_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **flush** construct, see Section 15.8.5.

19.5.2.18 ompt_callback_cancel_t

Summary

The **ompt_callback_cancel_t** type is used for callbacks that are dispatched for *cancellation*, *cancel* and *discarded-task* events.

Format

C / C++

```
typedef void (*ompt_callback_cancel_t) (
  ompt_data_t *task_data,
  int flags,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_cancel_t {
  ompt_id_t task_id;
  int flags;
  const void *codeptr_ra;
} ompt_record_cancel_t;
```

C / C++

Description of Arguments

The binding of the *task_data* argument is the task that encounters a **cancel** construct, a **cancellation point** construct, or a construct defined as having an implicit cancellation point.

The *flags* argument, defined by the **ompt_cancel_flag_t** enumeration type, indicates whether cancellation is activated by the current task, or detected as being activated by another task. The construct that is being canceled is also described in the *flags* argument. When several constructs are detected as being concurrently canceled, each corresponding bit in the argument will be set.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_cancel_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Cross References

- **omp_cancel_flag_t** enumeration type, see Section 19.4.4.26.

19.5.2.19 ompt_callback_device_initialize_t

Summary

The **ompt_callback_device_initialize_t** type is used for callbacks that initialize device tracing interfaces.

Format

C / C++

```
typedef void (*ompt_callback_device_initialize_t) (
  int device_num,
  const char *type,
  ompt_device_t *device,
  ompt_function_lookup_t lookup,
  const char *documentation
);
```

C / C++

Semantics

Registration of a callback with type signature **ompt_callback_device_initialize_t** for the **ompt_callback_device_initialize** event enables asynchronous collection of a trace for a device. The OpenMP implementation invokes this callback after OpenMP is initialized for the device but before execution of any OpenMP construct is started on the device.

Description of Arguments

The *device_num* argument identifies the logical device that is being initialized.

The *type* argument is a character string that indicates the type of the device. A device type string is a semicolon-separated character string that includes at a minimum the vendor and model name of the device. These names may be followed by a semicolon-separated sequence of properties that describe the hardware or software of the device.

The *device* argument is a pointer to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

The *lookup* argument points to a runtime callback that a tool must use to obtain pointers to runtime entry points in the device's OMPT tracing interface. If a device does not support tracing then *lookup* is *NULL*.

The *documentation* argument is a string that describes how to use any device-specific runtime entry points that can be obtained through the *lookup* argument. This documentation string may be a pointer to external documentation, or it may be inline descriptions that include names and type signatures for any device-specific interfaces that are available through the *lookup* argument along with descriptions of how to use these interface functions to control monitoring and analysis of device traces.

Constraints on Arguments

The *type* and *documentation* arguments must be immutable strings that are defined for the lifetime of program execution.

Effect

A device initializer must fulfill several duties. First, the *type* argument should be used to determine if any special knowledge about the hardware and/or software of a device is employed. Second, the *lookup* argument should be used to look up pointers to runtime entry points in the OMPT tracing interface for the device. Finally, these runtime entry points should be used to set up tracing for the device.

Initialization of tracing for a target device is described in Section 19.2.5.

Cross References

- **ompt_function_lookup_t** type, see Section 19.6.3.

19.5.2.20 ompt_callback_device_finalize_t

Summary

The **ompt_callback_device_initialize_t** type is used for callbacks that finalize device tracing interfaces.

Format

C / C++

```
typedef void (*ompt_callback_device_finalize_t) (
  int device_num
);
```

C / C++

Description of Arguments

The *device_num* argument identifies the logical device that is being finalized.

Semantics

A registered callback with type signature **ompt_callback_device_finalize_t** is dispatched for a device immediately prior to finalizing the device. Prior to dispatching a finalization callback for a device on which tracing is active, the OpenMP implementation stops tracing on the device and synchronously flushes all trace records for the device that have not yet been reported. These trace records are flushed through one or more buffer completion callbacks with type signature **ompt_callback_buffer_complete_t** as needed prior to the dispatch of the callback with type signature **ompt_callback_device_finalize_t**.

Cross References

- **ompt_callback_buffer_complete_t** callback type, see Section 19.5.2.24.

19.5.2.21 `ompt_callback_device_load_t`

Summary

The **`ompt_callback_device_load_t`** type is used for callbacks that the OpenMP runtime invokes to indicate that it has just loaded code onto the specified device.

Format

C / C++

```
typedef void (*ompt_callback_device_load_t) (
  int device_num,
  const char *filename,
  int64_t offset_in_file,
  void *vma_in_file,
  size_t bytes,
  void *host_addr,
  void *device_addr,
  uint64_t module_id
);
```

C / C++

Description of Arguments

The *device_num* argument specifies the device.

The *filename* argument indicates the name of a file in which the device code can be found. A *NULL* *filename* indicates that the code is not available in a file in the file system.

The *offset_in_file* argument indicates an offset into *filename* at which the code can be found. A value of -1 indicates that no offset is provided.

`ompt_addr_none` is defined as a pointer with the value ~0.

The *vma_in_file* argument indicates a virtual address in *filename* at which the code can be found. A value of **`ompt_addr_none`** indicates that a virtual address in the file is not available.

The *bytes* argument indicates the size of the device code object in bytes.

The *host_addr* argument indicates the address at which a copy of the device code is available in host memory. A value of **`ompt_addr_none`** indicates that a host code address is not available.

The *device_addr* argument indicates the address at which the device code has been loaded in device memory. A value of **`ompt_addr_none`** indicates that a device code address is not available.

The *module_id* argument is an identifier that is associated with the device code object.

Cross References

- Device directives, see Section 13.

19.5.2.22 `ompt_callback_device_unload_t`

Summary

The **`ompt_callback_device_unload_t`** type is used for callbacks that the OpenMP runtime invokes to indicate that it is about to unload code from the specified device.

Format

C / C++

```
typedef void (*ompt_callback_device_unload_t) (
  int device_num,
  uint64_t module_id
);
```

C / C++

Description of Arguments

The *device_num* argument specifies the device.

The *module_id* argument is an identifier that is associated with the device code object.

Cross References

- Device directives, see Section 13.

19.5.2.23 `ompt_callback_buffer_request_t`

Summary

The **`ompt_callback_buffer_request_t`** type is used for callbacks that are dispatched when a buffer to store event records for a device is requested.

Format

C / C++

```
typedef void (*ompt_callback_buffer_request_t) (
  int device_num,
  ompt_buffer_t **buffer,
  size_t *bytes
);
```

C / C++

Semantics

A callback with type signature **`ompt_callback_buffer_request_t`** requests a buffer to store trace records for the specified device. A buffer request callback may set **bytes* to 0 if it does not provide a buffer. If a callback sets **bytes* to 0, further recording of events for the device is disabled until the next invocation of **`ompt_start_trace`**. This action causes the device to drop future trace records until recording is restarted.

Description of Arguments

The *device_num* argument specifies the device.

The **buffer* argument points to a buffer where device events may be recorded. The **bytes* argument indicates the length of that buffer.

Cross References

- **ompt_buffer_t** type, see Section 19.4.4.7.

19.5.2.24 ompt_callback_buffer_complete_t

Summary

The **ompt_callback_buffer_complete_t** type is used for callbacks that are dispatched when devices will not record any more trace records in an event buffer and all records written to the buffer are valid.

Format

C / C++

```
typedef void (*ompt_callback_buffer_complete_t) (
  int device_num,
  ompt_buffer_t *buffer,
  size_t bytes,
  ompt_buffer_cursor_t begin,
  int buffer_owned
);
```

C / C++

Semantics

A callback with type signature **ompt_callback_buffer_complete_t** provides a buffer that contains trace records for the specified device. Typically, a tool will iterate through the records in the buffer and process them.

The OpenMP implementation makes these callbacks on a thread that is not an OpenMP primary or worker thread.

The callee may not delete the buffer if the *buffer_owned* argument is 0.

The buffer completion callback is not required to be *async signal safe*.

Description of Arguments

The *device_num* argument indicates the device for which the buffer contains events.

The *buffer* argument is the address of a buffer that was previously allocated by a *buffer request* callback.

The *bytes* argument indicates the full size of the buffer.

The *begin* argument is an opaque cursor that indicates the position of the beginning of the first record in the buffer.

The *buffer_owned* argument is 1 if the data to which the buffer points can be deleted by the callback and 0 otherwise. If multiple devices accumulate trace events into a single buffer, this callback may be invoked with a pointer to one or more trace records in a shared buffer with *buffer_owned* = 0. In this case, the callback may not delete the buffer.

Cross References

- **ompt_buffer_cursor_t** type, see Section 19.4.4.8.
- **ompt_buffer_t** type, see Section 19.4.4.7.

19.5.2.25 ompt_callback_target_data_op_emi_t and ompt_callback_target_data_op_t

Summary

The**ompt_callback_target_data_op_emi_t** and **ompt_callback_target_data_op_t** types are used for callbacks that are dispatched when a thread maps data to a device.

Format

C / C++

```
typedef void (*ompt_callback_target_data_op_emi_t) (
  ompt_scope_endpoint_t endpoint,
  ompt_data_t *target_task_data,
  ompt_data_t *target_data,
  ompt_id_t *host_op_id,
  ompt_target_data_op_t optype,
  void *src_addr,
  int src_device_num,
  void *dest_addr,
  int dest_device_num,
  size_t bytes,
  const void *codeptr_ra
);
```

```
typedef void (*ompt_callback_target_data_op_t) (
  ompt_id_t target_id,
  ompt_id_t host_op_id,
  ompt_target_data_op_t optype,
  void *src_addr,
  int src_device_num,
  void *dest_addr,
  int dest_device_num,
  size_t bytes,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_data_op_t {
  ompt_id_t host_op_id;
  ompt_target_data_op_t optype;
  void *src_addr;
  int src_device_num;
  void *dest_addr;
  int dest_device_num;
  size_t bytes;
  ompt_device_time_t end_time;
  const void *codeptr_ra;
} ompt_record_target_data_op_t;
```

C / C++

Semantics

A thread dispatches a registered **ompt_callback_target_data_op_emi** or **ompt_callback_target_data_op** callback when device memory is allocated or freed, as well as when data is copied to or from a device.

Note – An OpenMP implementation may aggregate program variables and data operations upon them. For instance, an OpenMP implementation may synthesize a composite to represent multiple scalars and then allocate, free, or copy this composite as a whole rather than performing data operations on each scalar individually. Thus, callbacks may not be dispatched as separate data operations on each variable.

Description of Arguments

The *endpoint* argument indicates that the callback signals the beginning or end of a scope.

The binding of the *target_task_data* argument is the target task region.

The binding of the *target_data* argument is the target region.

The *host_op_id* argument points to a tool controlled integer value, which identifies a data operation on a target device.

The *optype* argument indicates the kind of data operation.

The *src_addr* argument indicates the data address before the operation, where applicable.

The *src_device_num* argument indicates the source device number for the data operation, where applicable.

The *dest_addr* argument indicates the data address after the operation.

The *dest_device_num* argument indicates the destination device number for the data operation.

Whether in some operations *src_addr* or *dest_addr* may point to an intermediate buffer is implementation defined.

The *bytes* argument indicates the size of data.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_target_data_op_emi_t** or **ompt_callback_target_data_op_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Restrictions

Restrictions to the **ompt_callback_target_data_op_emi** and **ompt_callback_target_data_op** callbacks are as follows:

- These callbacks must not be registered at the same time.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_id_t** type, see Section 19.4.4.3.
- **ompt_scope_endpoint_t** type, see Section 19.4.4.11.
- **ompt_target_data_op_t** type, see Section 19.4.4.15.
- **map** clause, see Section 5.8.2.

19.5.2.26 `ompt_callback_target_emi_t` and `ompt_callback_target_t`

Summary

The **ompt_callback_target_emi_t** and **ompt_callback_target_t** types are used for callbacks that are dispatched when a thread begins to execute a device construct.

Format

C / C++

```c
typedef void (*ompt_callback_target_emi_t) (
  ompt_target_t kind,
  ompt_scope_endpoint_t endpoint,
  int device_num,
  ompt_data_t *task_data,
  ompt_data_t *target_task_data,
  ompt_data_t *target_data,
  const void *codeptr_ra
);
```

```c
typedef void (*ompt_callback_target_t) (
  ompt_target_t kind,
  ompt_scope_endpoint_t endpoint,
  int device_num,
  ompt_data_t *task_data,
  ompt_id_t target_id,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```c
typedef struct ompt_record_target_t {
  ompt_target_t kind;
  ompt_scope_endpoint_t endpoint;
  int device_num;
  ompt_id_t task_id;
  ompt_id_t target_id;
  const void *codeptr_ra;
} ompt_record_target_t;
```

C / C++

Description of Arguments

The *kind* argument indicates the kind of target region.

The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a scope.

The *device_num* argument indicates the device number of the device that will execute the target region.

The binding of the *task_data* argument is the encountering task.

The binding of the *target_task_data* argument is the target task region. If a target region has no target task or if the target task is merged, this argument is **NULL**.

The binding of the *target_data* argument is the target region.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_target_emi_t** or **ompt_callback_target_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Restrictions

Restrictions to the **ompt_callback_target_emi** and **ompt_callback_target** callbacks are as follows:

- These callbacks must not be registered at the same time.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_id_t** type, see Section 19.4.4.3.
- **ompt_scope_endpoint_t** type, see Section 19.4.4.11.
- **ompt_target_t** type, see Section 19.4.4.21.
- **target** construct, see Section 13.8.
- **target data** construct, see Section 13.5.
- **target enter data** construct, see Section 13.6.
- **target exit data** construct, see Section 13.7.
- **target update** construct, see Section 13.9.

19.5.2.27 `ompt_callback_target_map_emi_t` and `ompt_callback_target_map_t`

Summary

The **ompt_callback_target_map_emi_t** and **ompt_callback_target_map_t** types are used for callbacks that are dispatched to indicate data mapping relationships.

Format

C / C++

```
typedef void (*ompt_callback_target_map_emi_t) (
  ompt_data_t *target_data,
  unsigned int nitems,
  void **host_addr,
  void **device_addr,
  size_t *bytes,
  unsigned int *mapping_flags,
  const void *codeptr_ra
);
```

```
typedef void (*ompt_callback_target_map_t) (
  ompt_id_t target_id,
  unsigned int nitems,
  void **host_addr,
  void **device_addr,
  size_t *bytes,
  unsigned int *mapping_flags,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_map_t {
  ompt_id_t target_id;
  unsigned int nitems;
  void **host_addr;
  void **device_addr;
  size_t *bytes;
  unsigned int *mapping_flags;
  const void *codeptr_ra;
} ompt_record_target_map_t;
```

C / C++

Semantics

An instance of a **target**, **target data**, **target enter data**, or **target exit data** construct may contain one or more **map** clauses. An OpenMP implementation may report the set of mappings associated with **map** clauses for a construct with a single **ompt_callback_target_map_emi** or **ompt_callback_target_map** callback to report the effect of all mappings or multiple **ompt_callback_target_map_emi** or **ompt_callback_target_map** callbacks with each reporting a subset of the mappings. Furthermore, an OpenMP implementation may omit mappings that it determines are unnecessary. If an OpenMP implementation issues multiple **ompt_callback_target_map_emi** or **ompt_callback_target_map** callbacks, these callbacks may be interleaved with **ompt_callback_target_data_op_emi** or **ompt_callback_target_data_op** callbacks used to report data operations associated with the mappings.

Description of Arguments

The binding of the *target_data* argument is the target region.

The *nitems* argument indicates the number of data mappings that this callback reports.

The *host_addr* argument indicates an array of host data addresses.

The *device_addr* argument indicates an array of device data addresses.

The *bytes* argument indicates an array of sizes of data.

The *mapping_flags* argument indicates the kind of mapping operations, which may result from explicit **map** clauses or the implicit data-mapping rules defined in Section 5.8. Flags for the mapping operations include one or more values specified by the **ompt_target_map_flag_t** type.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_target_map_t** or **ompt_callback_target_map_emi_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Restrictions

Restrictions to the **ompt_callback_target_data_map_emi** and **ompt_callback_target_data_map** callbacks are as follows:

- These callbacks must not be registered at the same time.

Cross References

- **ompt_callback_target_data_op_emi_t** or **ompt_callback_target_data_op_t** callback type, see Section 19.5.2.25.
- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_id_t** type, see Section 19.4.4.3.
- **ompt_target_map_flag_t** type, see Section 19.4.4.23.
- **target** construct, see Section 13.8.
- **target data** construct, see Section 13.5.
- **target enter data** construct, see Section 13.6.
- **target exit data** construct, see Section 13.7.

19.5.2.28 ompt_callback_target_submit_emi_t and ompt_callback_target_submit_t

Summary

The **ompt_callback_target_submit_emi_t** and **ompt_callback_target_submit_t** types are used for callbacks that are dispatched before and after the host initiates creation of an initial task on a device.

Format

C / C++

```
typedef void (*ompt_callback_target_submit_emi_t) (
  ompt_scope_endpoint_t endpoint,
  ompt_data_t *target_data,
  ompt_id_t *host_op_id,
  unsigned int requested_num_teams
);
```

```
typedef void (*ompt_callback_target_submit_t) (
  ompt_id_t target_id,
  ompt_id_t host_op_id,
  unsigned int requested_num_teams
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_kernel_t {
  ompt_id_t host_op_id;
  unsigned int requested_num_teams;
  unsigned int granted_num_teams;
  ompt_device_time_t end_time;
} ompt_record_target_kernel_t;
```

C / C++

Semantics

A thread dispatches a registered **ompt_callback_target_submit_emi** or **ompt_callback_target_submit** callback on the host before and after a target task initiates creation of an initial task on a device.

Description of Arguments

The *endpoint* argument indicates that the callback signals the beginning or end of a scope.

The binding of the *target_data* argument is the target region.

The *host_op_id* argument points to a tool controlled integer value, which identifies an initial task on a target device.

The *requested_num_teams* argument is the number of teams that the host requested to execute the kernel. The actual number of teams that execute the kernel may be smaller and generally will not be known until the kernel begins to execute on the device.

If **ompt_set_trace_ompt** has configured the device to trace kernel execution then the device will log a **ompt_record_target_kernel_t** record in a trace. The fields in the record are as follows:

- The *host_op_id* field contains a tool-controlled identifier that can be used to correlate a **ompt_record_target_kernel_t** record with its associated **ompt_callback_target_submit_emi** or **ompt_callback_target_submit** callback on the host;
- The *requested_num_teams* field contains the number of teams that the host requested to execute the kernel;
- The *granted_num_teams* field contains the number of teams that the device actually used to execute the kernel;
- The time when the initial task began execution on the device is recorded in the *time* field of an enclosing **ompt_record_t** structure; and
- The time when the initial task completed execution on the device is recorded in the *end_time* field.

Restrictions

Restrictions to the **ompt_callback_target_submit_emi** and **ompt_callback_target_submit** callbacks are as follows:

- These callbacks must not be registered at the same time.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompt_id_t** type, see Section 19.4.4.3.
- **ompt_scope_endpoint_t** type, see Section 19.4.4.11.
- **target** construct, see Section 13.8.

19.5.2.29 ompt_callback_control_tool_t

Summary

The **ompt_callback_control_tool_t** type is used for callbacks that dispatch *tool-control* events.

Format

C / C++

```
typedef int (*ompt_callback_control_tool_t) (
  uint64_t command,
  uint64_t modifier,
  void *arg,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_control_tool_t {
  uint64_t command;
  uint64_t modifier;
  const void *codeptr_ra;
} ompt_record_control_tool_t;
```

C / C++

Semantics

Callbacks with type signature **ompt_callback_control_tool_t** may return any non-negative value, which will be returned to the application as the return value of the **omp_control_tool** call that triggered the callback.

Description of Arguments

The *command* argument passes a command from an application to a tool. Standard values for *command* are defined by **omp_control_tool_t** in Section 18.14.

The *modifier* argument passes a command modifier from an application to a tool.

The *command* and *modifier* arguments may have tool-specific values. Tools must ignore *command* values that they are not designed to handle.

The *arg* argument is a void pointer that enables a tool and an application to exchange arbitrary state. The *arg* argument may be *NULL*.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_control_tool_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Constraints on Arguments

Tool-specific values for *command* must be ≥ 64.

Cross References

- Tool control routine and types, see Section 18.14.

19.5.2.30 ompt_callback_error_t

Summary

The **ompt_callback_error_t** type is used for callbacks that dispatch *runtime-error* events.

Format

C / C++

```
typedef void (*ompt_callback_error_t) (
  ompt_severity_t severity,
  const char *message,
  size_t length,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_error_t {
  ompt_severity_t severity;
  const char *message;
  size_t length;
  const void *codeptr_ra;
} ompt_record_error_t;
```

C / C++

Semantics

A thread dispatches a registered **ompt_callback_error_t** callback when an **error** directive is encountered for which the **at(execution)** clause is specified.

Description of Arguments

The *severity* argument passes the specified severity level.

The *message* argument passes the string from the **message** clause.

The *length* argument provides the length of the string.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_error_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be *NULL*.

Cross References

- **ompt_severity_t** enumeration type, see Section 19.4.4.25.
- **error** directive, see Section 8.5.

19.6 OMPT Runtime Entry Points for Tools

OMPT supports two principal sets of runtime entry points for tools. One set of runtime entry points enables a tool to register callbacks for OpenMP events and to inspect the state of an OpenMP thread while executing in a tool callback or a signal handler. The second set of runtime entry points enables a tool to trace activities on a device. When directed by the tracing interface, an OpenMP implementation will trace activities on a device, collect buffers of trace records, and invoke callbacks on the host to process these records. OMPT runtime entry points should not be global symbols since tools cannot rely on the visibility of such symbols.

OMPT also supports runtime entry points for two classes of lookup routines. The first class of lookup routines contains a single member: a routine that returns runtime entry points in the OMPT callback interface. The second class of lookup routines includes a unique lookup routine for each kind of device that can return runtime entry points in a device's OMPT tracing interface.

The **omp-tools.h** C/C++ header file provides the definitions of the types that are specified throughout this subsection.

Binding

The binding thread set for each of the entry points in this section is the encountering thread unless otherwise specified. The binding task set is the task executing on the encountering thread.

Restrictions

Restrictions on OMPT runtime entry points are as follows:

- OMPT runtime entry points must not be called from a signal handler on a native thread before a *native-thread-begin* or after a *native-thread-end* event.
- OMPT device runtime entry points must not be called after a *device-finalize* event for that device.

19.6.1 Entry Points in the OMPT Callback Interface

Entry points in the OMPT callback interface enable a tool to register callbacks for OpenMP events and to inspect the state of an OpenMP thread while executing in a tool callback or a signal handler. Pointers to these runtime entry points are obtained through the lookup function that is provided through the OMPT initializer.

19.6.1.1 `ompt_enumerate_states_t`

Summary

The **ompt_enumerate_states_t** type is the type signature of the **ompt_enumerate_states** runtime entry point, which enumerates the thread states that an OpenMP implementation supports.

Format

C / C++

```
typedef int (*ompt_enumerate_states_t) (
  int current_state,
  int *next_state,
  const char **next_state_name
);
```

C / C++

Semantics

An OpenMP implementation may support only a subset of the states defined by the **ompt_state_t** enumeration type. An OpenMP implementation may also support implementation-specific states. The **ompt_enumerate_states** runtime entry point, which has type signature **ompt_enumerate_states_t**, enables a tool to enumerate the supported thread states.

When a supported thread state is passed as *current_state*, the runtime entry point assigns the next thread state in the enumeration to the variable passed by reference in *next_state* and assigns the name associated with that state to the character pointer passed by reference in *next_state_name*.

Whenever one or more states are left in the enumeration, the **ompt_enumerate_states** runtime entry point returns 1. When the last state in the enumeration is passed as *current_state*, **ompt_enumerate_states** returns 0, which indicates that the enumeration is complete.

Description of Arguments

The *current_state* argument must be a thread state that the OpenMP implementation supports. To begin enumerating the supported states, a tool should pass **ompt_state_undefined** as *current_state*. Subsequent invocations of **ompt_enumerate_states** should pass the value assigned to the variable that was passed by reference in *next_state* to the previous call.

The value **ompt_state_undefined** is reserved to indicate an invalid thread state. **ompt_state_undefined** is defined as an integer with the value 0.

The *next_state* argument is a pointer to an integer in which **ompt_enumerate_states** returns the value of the next state in the enumeration.

The *next_state_name* argument is a pointer to a character string pointer through which **ompt_enumerate_states** returns a string that describes the next state.

Constraints on Arguments

Any string returned through the *next_state_name* argument must be immutable and defined for the lifetime of program execution.

Cross References

- **ompt_state_t** type, see Section 19.4.4.28.

19.6.1.2 ompt_enumerate_mutex_impls_t

Summary

The **ompt_enumerate_mutex_impls_t** type is the type signature of the **ompt_enumerate_mutex_impls** runtime entry point, which enumerates the kinds of mutual exclusion implementations that an OpenMP implementation employs.

Format

C / C++

```
typedef int (*ompt_enumerate_mutex_impls_t) (
  int current_impl,
  int *next_impl,
  const char **next_impl_name
);
```

C / C++

Semantics

Mutual exclusion for locks, **critical** sections, and **atomic** regions may be implemented in several ways. The **ompt_enumerate_mutex_impls** runtime entry point, which has type signature **ompt_enumerate_mutex_impls_t**, enables a tool to enumerate the supported mutual exclusion implementations.

When a supported mutex implementation is passed as *current_impl*, the runtime entry point assigns the next mutex implementation in the enumeration to the variable passed by reference in *next_impl* and assigns the name associated with that mutex implementation to the character pointer passed by reference in *next_impl_name*.

Whenever one or more mutex implementations are left in the enumeration, the **ompt_enumerate_mutex_impls** runtime entry point returns 1. When the last mutex implementation in the enumeration is passed as *current_impl*, the runtime entry point returns 0, which indicates that the enumeration is complete.

Description of Arguments

The *current_impl* argument must be a mutex implementation that an OpenMP implementation supports. To begin enumerating the supported mutex implementations, a tool should pass **ompt_mutex_impl_none** as *current_impl*. Subsequent invocations of **ompt_enumerate_mutex_impls** should pass the value assigned to the variable that was passed in *next_impl* to the previous call.

The value **ompt_mutex_impl_none** is reserved to indicate an invalid mutex implementation. **ompt_mutex_impl_none** is defined as an integer with the value 0.

The *next_impl* argument is a pointer to an integer in which **ompt_enumerate_mutex_impls** returns the value of the next mutex implementation in the enumeration.

The *next_impl_name* argument is a pointer to a character string pointer in which **ompt_enumerate_mutex_impls** returns a string that describes the next mutex implementation.

Constraints on Arguments

Any string returned through the *next_impl_name* argument must be immutable and defined for the lifetime of a program execution.

Cross References

- **ompt_mutex_t** type, see Section 19.4.4.17.

19.6.1.3 ompt_set_callback_t

Summary

The **ompt_set_callback_t** type is the type signature of the **ompt_set_callback** runtime entry point, which registers a pointer to a tool callback that an OpenMP implementation invokes when a host OpenMP event occurs.

Format

C / C++

```
typedef ompt_set_result_t (*ompt_set_callback_t) (
  ompt_callbacks_t event,
  ompt_callback_t callback
);
```

C / C++

Semantics

OpenMP implementations can use callbacks to indicate the occurrence of events during the execution of an OpenMP program. The **ompt_set_callback** runtime entry point, which has type signature **ompt_set_callback_t**, registers a callback for an OpenMP event on the current device, The return value of **ompt_set_callback** indicates the outcome of registering the callback.

Description of Arguments

The *event* argument indicates the event for which the callback is being registered.

The *callback* argument is a tool callback function. If *callback* is *NULL* then callbacks associated with *event* are disabled. If callbacks are successfully disabled then **ompt_set_always** is returned.

Constraints on Arguments

When a tool registers a callback for an event, the type signature for the callback must match the type signature appropriate for the event.

Restrictions

Restrictions on the **ompt_set_callback** runtime entry point are as follows:

- The entry point must not return **ompt_set_impossible**.

Cross References

- Monitoring activity on the host with OMPT, see Section 19.2.4.
- **ompt_callback_t** type, see Section 19.4.4.1.
- **ompt_callbacks_t** enumeration type, see Section 19.4.2.
- **ompt_get_callback_t** host callback type signature, see Section 19.6.1.4.
- **ompt_set_result_t** type, see Section 19.4.4.2.

19.6.1.4 ompt_get_callback_t

Summary

The **ompt_get_callback_t** type is the type signature of the **ompt_get_callback** runtime entry point, which retrieves a pointer to a registered tool callback routine (if any) that an OpenMP implementation invokes when a host OpenMP event occurs.

Format

C / C++

```
typedef int (*ompt_get_callback_t) (
  ompt_callbacks_t event,
  ompt_callback_t *callback
);
```

C / C++

Semantics

The **ompt_get_callback** runtime entry point, which has type signature **ompt_get_callback_t**, retrieves a pointer to the tool callback that an OpenMP implementation may invoke when a host OpenMP event occurs. If a non-null tool callback is registered for the specified event, the pointer to the tool callback is assigned to the variable passed by reference in *callback* and **ompt_get_callback** returns 1; otherwise, it returns 0. If **ompt_get_callback** returns 0, the value of the variable passed by reference as *callback* is undefined.

Description of Arguments

The *event* argument indicates the event for which the callback would be invoked.

The *callback* argument returns a pointer to the callback associated with *event*.

Constraints on Arguments

The *callback* argument cannot be *NULL* and must point to valid storage.

Cross References

- **ompt_callback_t** type, see Section 19.4.4.1.
- **ompt_callbacks_t** enumeration type, see Section 19.4.2.
- **ompt_set_callback_t** type signature, see Section 19.6.1.3.

19.6.1.5 ompt_get_thread_data_t

Summary

The **ompt_get_thread_data_t** type is the type signature of the **ompt_get_thread_data** runtime entry point, which returns the address of the thread data object for the current thread.

Format

C / C++

```
typedef ompt_data_t *(*ompt_get_thread_data_t) (void);
```

C / C++

Semantics

Each OpenMP thread can have an associated thread data object of type **ompt_data_t**. The **ompt_get_thread_data** runtime entry point, which has type signature **ompt_get_thread_data_t**, retrieves a pointer to the thread data object, if any, that is associated with the current thread. A tool may use a pointer to an OpenMP thread's data object that **ompt_get_thread_data** retrieves to inspect or to modify the value of the data object. When an OpenMP thread is created, its data object is initialized with value **ompt_data_none**.

This runtime entry point is *async signal safe*.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.

19.6.1.6 ompt_get_num_procs_t

Summary

The **ompt_get_num_procs_t** type is the type signature of the **ompt_get_num_procs** runtime entry point, which returns the number of processors currently available to the execution environment on the host device.

Format

C / C++

```
typedef int (*ompt_get_num_procs_t) (void);
```

C / C++

Binding

The binding thread set is all threads on the host device.

Semantics

The **ompt_get_num_procs** runtime entry point, which has type signature **ompt_get_num_procs_t**, returns the number of processors that are available on the host device at the time the routine is called. This value may change between the time that it is determined and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation.

This runtime entry point is *async signal safe*.

19.6.1.7 ompt_get_num_places_t

Summary

The **ompt_get_num_places_t** type is the type signature of the **ompt_get_num_places** runtime entry point, which returns the number of places currently available to the execution environment in the place list.

Format

C / C++

```
typedef int (*ompt_get_num_places_t) (void);
```

C / C++

Binding

The binding thread set is all threads on a device.

Semantics

The **ompt_get_num_places** runtime entry point, which has type signature **ompt_get_num_places_t**, returns the number of places in the place list. This value is equivalent to the number of places in the *place-partition-var* ICV in the execution environment of the initial task.

This runtime entry point is *async signal safe*.

Cross References

- *place-partition-var* ICV, see Section 2.
- **OMP_PLACES** environment variable, see Section 21.1.6.

19.6.1.8 ompt_get_place_proc_ids_t

Summary

The **ompt_get_place_procs_ids_t** type is the type signature of the **ompt_get_num_place_procs_ids** runtime entry point, which returns the numerical identifiers of the processors that are available to the execution environment in the specified place.

Format

C / C++

```
typedef int (*ompt_get_place_proc_ids_t) (
  int place_num,
  int ids_size,
  int *ids
);
```

C / C++

Binding

The binding thread set is all threads on a device.

Semantics

The **ompt_get_place_proc_ids** runtime entry point, which has type signature **ompt_get_place_proc_ids_t**, returns the numerical identifiers of each processor that is associated with the specified place. These numerical identifiers are non-negative, and their meaning is implementation defined.

Description of Arguments

The *place_num* argument specifies the place that is being queried.

The *ids* argument is an array in which the routine can return a vector of processor identifiers in the specified place.

The *ids_size* argument indicates the size of the result array that is specified by *ids*.

Effect

If the *ids* array of size *ids_size* is large enough to contain all identifiers then they are returned in *ids* and their order in the array is implementation defined. Otherwise, if the *ids* array is too small, the values in *ids* when the function returns are unspecified. The routine always returns the number of numerical identifiers of the processors that are available to the execution environment in the specified place.

19.6.1.9 `ompt_get_place_num_t`

Summary

The **`ompt_get_place_num_t`** type is the type signature of the **`ompt_get_place_num`** runtime entry point, which returns the place number of the place to which the current thread is bound.

Format

C / C++

```
typedef int (*ompt_get_place_num_t) (void);
```

C / C++

Semantics

When the current thread is bound to a place, **`ompt_get_place_num`** returns the place number associated with the thread. The returned value is between 0 and one less than the value returned by **`ompt_get_num_places`**, inclusive. When the current thread is not bound to a place, the routine returns -1.

This runtime entry point is *async signal safe*.

19.6.1.10 `ompt_get_partition_place_nums_t`

Summary

The **`ompt_get_partition_place_nums_t`** type is the type signature of the **`ompt_get_partition_place_nums`** runtime entry point, which returns a list of place numbers that correspond to the places in the *place-partition-var* ICV of the innermost implicit task.

Format

C / C++

```
typedef int (*ompt_get_partition_place_nums_t) (
  int place_nums_size,
  int *place_nums
);
```

C / C++

Semantics

The **`ompt_get_partition_place_nums`** runtime entry point, which has type signature **`ompt_get_partition_place_nums_t`**, returns a list of place numbers that correspond to the places in the *place-partition-var* ICV of the innermost implicit task.

This runtime entry point is *async signal safe*.

Description of Arguments

The *place_nums* argument is an array in which the routine can return a vector of place identifiers.

The *place_nums_size* argument indicates the size of the result array that the *place_nums* argument specifies.

Effect

If the *place_nums* array of size *place_nums_size* is large enough to contain all identifiers then they are returned in *place_nums* and their order in the array is implementation defined. Otherwise, if the *place_nums* array is too small, the values in *place_nums* when the function returns are unspecified. The routine always returns the number of places in the *place-partition-var* ICV of the innermost implicit task.

Cross References

- *place-partition-var* ICV, see Section 2.
- **`OMP_PLACES`** environment variable, see Section 21.1.6.

19.6.1.11 `ompt_get_proc_id_t`

Summary

The **`ompt_get_proc_id_t`** type is the type signature of the **`ompt_get_proc_id`** runtime entry point, which returns the numerical identifier of the processor of the current thread.

Format

C / C++

```
typedef int (*ompt_get_proc_id_t) (void);
```

C / C++

Semantics

The **`ompt_get_proc_id`** runtime entry point, which has type signature **`ompt_get_proc_id_t`**, returns the numerical identifier of the processor of the current thread. A defined numerical identifier is non-negative, and its meaning is implementation defined. A negative number indicates a failure to retrieve the numerical identifier.

This runtime entry point is *async signal safe*.

19.6.1.12 `ompt_get_state_t`

Summary

The **`ompt_get_state_t`** type is the type signature of the **`ompt_get_state`** runtime entry point, which returns the state and the wait identifier of the current thread.

Format

C / C++

```
typedef int (*ompt_get_state_t) (
  ompt_wait_id_t *wait_id
);
```

C / C++

Semantics

Each OpenMP thread has an associated state and a wait identifier. If a thread's state indicates that the thread is waiting for mutual exclusion then its wait identifier contains an opaque handle that indicates the data object upon which the thread is waiting. The **ompt_get_state** runtime entry point, which has type signature **ompt_get_state_t**, retrieves the state and wait identifier of the current thread. The returned value may be any one of the states predefined by **ompt_state_t** or a value that represents an implementation-specific state. The tool may obtain a string representation for each state with the **ompt_enumerate_states** function.

If the returned state indicates that the thread is waiting for a lock, nest lock, **critical** region, **atomic** region, or **ordered** region then the value of the thread's wait identifier is assigned to a non-null wait identifier passed as the *wait_id* argument.

This runtime entry point is *async signal safe*.

Description of Arguments

The *wait_id* argument is a pointer to an opaque handle that is available to receive the value of the wait identifier of the thread. If *wait_id* is not *NULL* then the entry point assigns the value of the wait identifier of the thread to the object to which *wait_id* points. If the returned state is not one of the specified wait states then the value of opaque object to which *wait_id* points is undefined after the call.

Constraints on Arguments

The argument passed to the entry point must be a reference to a variable of the specified type or *NULL*.

Cross References

- **ompt_enumerate_states_t** type, see Section 19.6.1.1.
- **ompt_state_t** type, see Section 19.4.4.28.
- **ompt_wait_id_t** type, see Section 19.4.4.31.

19.6.1.13 ompt_get_parallel_info_t

Summary

The **ompt_get_parallel_info_t** type is the type signature of the **ompt_get_parallel_info** runtime entry point, which returns information about the parallel region, if any, at the specified ancestor level for the current execution context.

Format

C / C++

```
typedef int (*ompt_get_parallel_info_t) (
  int ancestor_level,
  ompt_data_t **parallel_data,
  int *team_size
);
```

C / C++

Semantics

During execution, an OpenMP program may employ nested parallel regions. The **ompt_get_parallel_info** runtime entry point, which has type signature **ompt_get_parallel_info_t**, retrieves information, about the current parallel region and any enclosing parallel regions for the current execution context. The entry point returns 2 if a parallel region exists at the specified ancestor level and the information is available, 1 if a parallel region exists at the specified ancestor level but the information is currently unavailable, and 0 otherwise.

A tool may use the pointer to the data object of a parallel region that it obtains from this runtime entry point to inspect or to modify the value of the data object. When a parallel region is created, its data object will be initialized with the value **ompt_data_none**.

This runtime entry point is *async signal safe*.

Between a *parallel-begin* event and an *implicit-task-begin* event, a call to **ompt_get_parallel_info(0,...)** may return information about the outer parallel team, the new parallel team or an inconsistent state.

If a thread is in the state **ompt_state_wait_barrier_implicit_parallel** then a call to **ompt_get_parallel_info** may return a pointer to a copy of the specified parallel region's *parallel_data* rather than a pointer to the data word for the region itself. This convention enables the primary thread for a parallel region to free storage for the region immediately after the region ends, yet avoid having some other thread in the team that is executing the region potentially reference the *parallel_data* object for the region after it has been freed.

Description of Arguments

The *ancestor_level* argument specifies the parallel region of interest by its ancestor level. Ancestor level 0 refers to the innermost parallel region; information about enclosing parallel regions may be obtained using larger values for *ancestor_level*.

The *parallel_data* argument returns the parallel data if the argument is not *NULL*.

The *team_size* argument returns the team size if the argument is not *NULL*.

Effect

If the runtime entry point returns 0 or 1, no argument is modified. Otherwise, **`ompt_get_parallel_info`** has the following effects:

- If a non-null value was passed for *parallel_data*, the value returned in *parallel_data* is a pointer to a data word that is associated with the parallel region at the specified level; and
- If a non-null value was passed for *team_size*, the value returned in the integer to which *team_size* point is the number of threads in the team that is associated with the parallel region.

Constraints on Arguments

While argument *ancestor_level* is passed by value, all other arguments to the entry point must be pointers to variables of the specified types or *NULL*.

Cross References

- **`ompt_data_t`** type, see Section 19.4.4.4.

19.6.1.14 `ompt_get_task_info_t`

Summary

The **`ompt_get_task_info_t`** type is the type signature of the **`ompt_get_task_info`** runtime entry point, which returns information about the task, if any, at the specified ancestor level in the current execution context.

Format

C / C++

```
typedef int (*ompt_get_task_info_t) (
  int ancestor_level,
  int *flags,
  ompt_data_t **task_data,
  ompt_frame_t **task_frame,
  ompt_data_t **parallel_data,
  int *thread_num
);
```

C / C++

Semantics

During execution, an OpenMP thread may be executing an OpenMP task. Additionally, the stack of the thread may contain procedure frames that are associated with suspended OpenMP tasks or OpenMP runtime system routines. To obtain information about any task on the stack of the current thread, a tool uses the **`ompt_get_task_info`** runtime entry point, which has type signature **`ompt_get_task_info_t`**.

Ancestor level 0 refers to the active task; information about other tasks with associated frames present on the stack in the current execution context may be queried at higher ancestor levels.

The **`ompt_get_task_info`** runtime entry point returns 2 if a task region exists at the specified ancestor level and the information is available, 1 if a task region exists at the specified ancestor level but the information is currently unavailable, and 0 otherwise.

If a task exists at the specified ancestor level and the information is available then information is returned in the variables passed by reference to the entry point. If no task region exists at the specified ancestor level or the information is unavailable then the values of variables passed by reference to the entry point are undefined when **`ompt_get_task_info`** returns.

A tool may use a pointer to a data object for a task or parallel region that it obtains from **`ompt_get_task_info`** to inspect or to modify the value of the data object. When either a parallel region or a task region is created, its data object will be initialized with the value **`ompt_data_none`**.

This runtime entry point is *async signal safe*.

Description of Arguments

The *ancestor_level* argument specifies the task region of interest by its ancestor level. Ancestor level 0 refers to the active task; information about ancestor tasks found in the current execution context may be queried at higher ancestor levels.

The *flags* argument returns the task type if the argument is not *NULL*.

The *task_data* argument returns the task data if the argument is not *NULL*.

The *task_frame* argument returns the task frame pointer if the argument is not *NULL*.

The *parallel_data* argument returns the parallel data if the argument is not *NULL*.

The *thread_num* argument returns the thread number if the argument is not *NULL*.

Effect

If the runtime entry point returns 0 or 1, no argument is modified. Otherwise, **`ompt_get_task_info`** has the following effects:

- If a non-null value was passed for *flags* then the value returned in the integer to which *flags* points represents the type of the task at the specified level; possible task types include initial, implicit, explicit, and target tasks;

- If a non-null value was passed for *task_data* then the value that is returned in the object to which it points is a pointer to a data word that is associated with the task at the specified level;
- If a non-null value was passed for *task_frame* then the value that is returned in the object to which *task_frame* points is a pointer to the **`ompt_frame_t`** structure that is associated with the task at the specified level;
- If a non-null value was passed for *parallel_data* then the value that is returned in the object to which *parallel_data* points is a pointer to a data word that is associated with the parallel region that contains the task at the specified level or, if the task at the specified level is an initial task, *NULL*; and
- If a non-null value was passed for *thread_num*, then the value that is returned in the object to which *thread_num* points indicates the number of the thread in the parallel region that is executing the task at the specified level.

Constraints on Arguments

While argument *ancestor_level* is passed by value, all other arguments to **`ompt_get_task_info`** must be pointers to variables of the specified types or *NULL*.

Cross References

- **`ompt_data_t`** type, see Section 19.4.4.4.
- **`ompt_frame_t`** type, see Section 19.4.4.29.
- **`ompt_task_flag_t`** type, see Section 19.4.4.19.

19.6.1.15 `ompt_get_task_memory_t`

Summary

The **`ompt_get_task_memory_t`** type is the type signature of the **`ompt_get_task_memory`** runtime entry point, which returns information about memory ranges that are associated with the task.

Format

C / C++

```
typedef int (*ompt_get_task_memory_t)(
  void **addr,
  size_t *size,
  int block
);
```

C / C++

Semantics

During execution, an OpenMP thread may be executing an OpenMP task. The OpenMP implementation must preserve the data environment from the creation of the task for the execution of the task. The **ompt_get_task_memory** runtime entry point, which has type signature **ompt_get_task_memory_t**, provides information about the memory ranges used to store the data environment for the current task.

Multiple memory ranges may be used to store these data. The *block* argument supports iteration over these memory ranges.

The **ompt_get_task_memory** runtime entry point returns 1 if more memory ranges are available, and 0 otherwise. If no memory is used for a task, *size* is set to 0. In this case, addr is unspecified.

This runtime entry point is *async signal safe*.

Description of Arguments

The *addr* argument is a pointer to a void pointer return value to provide the start address of a memory block.

The *size* argument is a pointer to a size type return value to provide the size of the memory block.

The *block* argument is an integer value to specify the memory block of interest.

19.6.1.16 ompt_get_target_info_t

Summary

The **ompt_get_target_info_t** type is the type signature of the **ompt_get_target_info** runtime entry point, which returns identifiers that specify a thread's current **target** region and target operation ID, if any.

Format

C / C++

```
typedef int (*ompt_get_target_info_t) (
  uint64_t *device_num,
  ompt_id_t *target_id,
  ompt_id_t *host_op_id
);
```

C / C++

Semantics

The **ompt_get_target_info** entry point, which has type signature **ompt_get_target_info_t**, returns 1 if the current thread is in a **target** region and 0 otherwise. If the entry point returns 0 then the values of the variables passed by reference as its arguments are undefined.

If the current thread is in a **target** region then **ompt_get_target_info** returns information about the current device, active **target** region, and active host operation, if any.

This runtime entry point is *async signal safe*.

Description of Arguments

The *device_num* argument returns the device number if the current thread is in a **target** region.

The *target_id* argument returns the **target** region identifier if the current thread is in a **target** region.

If the current thread is in the process of initiating an operation on a target device (for example, copying data to or from an accelerator or launching a kernel), then *host_op_id* returns the identifier for the operation; otherwise, *host_op_id* returns **ompt_id_none**.

Constraints on Arguments

Arguments passed to the entry point must be valid references to variables of the specified types.

Cross References

- **ompt_id_t** type, see Section 19.4.4.3.

19.6.1.17 ompt_get_num_devices_t

Summary

The **ompt_get_num_devices_t** type is the type signature of the **ompt_get_num_devices** runtime entry point, which returns the number of available devices.

Format

C / C++

```
typedef int (*ompt_get_num_devices_t) (void);
```

C / C++

Semantics

The **ompt_get_num_devices** runtime entry point, which has type signature **ompt_get_num_devices_t**, returns the number of devices available to an OpenMP program.

This runtime entry point is *async signal safe*.

19.6.1.18 `ompt_get_unique_id_t`

Summary

The **`ompt_get_unique_id_t`** type is the type signature of the **`ompt_get_unique_id`** runtime entry point, which returns a unique number.

Format

C / C++

```
typedef uint64_t (*ompt_get_unique_id_t) (void);
```

C / C++

Semantics

The **`ompt_get_unique_id`** runtime entry point, which has type signature **`ompt_get_unique_id_t`**, returns a number that is unique for the duration of an OpenMP program. Successive invocations may not result in consecutive or even increasing numbers.

This runtime entry point is *async signal safe*.

19.6.1.19 `ompt_finalize_tool_t`

Summary

The **`ompt_finalize_tool_t`** type is the type signature of the **`ompt_finalize_tool`** runtime entry point, which enables a tool to finalize itself.

Format

C / C++

```
typedef void (*ompt_finalize_tool_t) (void);
```

C / C++

Semantics

A tool may detect that the execution of an OpenMP program is ending before the OpenMP implementation does. To facilitate clean termination of the tool, the tool may invoke the **`ompt_finalize_tool`** runtime entry point, which has type signature **`ompt_finalize_tool_t`**. Upon completion of **`ompt_finalize_tool`**, no OMPT callbacks are dispatched.

Effect

The **`ompt_finalize_tool`** routine detaches the tool from the runtime, unregisters all callbacks and invalidates all OMPT entry points passed to the tool in the *lookup-function*. Upon completion of **`ompt_finalize_tool`**, no further callbacks will be issued on any thread.

Before the callbacks are unregistered, the OpenMP runtime should attempt to dispatch all outstanding registered callbacks as well as the callbacks that would be encountered during shutdown of the runtime, if possible in the current execution context.

19.6.2 Entry Points in the OMPT Device Tracing Interface

The runtime entry points with type signatures of the types that are specified in this section enable a tool to trace activities on a device.

19.6.2.1 `ompt_get_device_num_procs_t`

Summary

The **`ompt_get_device_num_procs_t`** type is the type signature of the **`ompt_get_device_num_procs`** runtime entry point, which returns the number of processors currently available to the execution environment on the specified device.

Format

C / C++

```
typedef int (*ompt_get_device_num_procs_t) (
  ompt_device_t *device
);
```

C / C++

Semantics

The **`ompt_get_device_num_procs`** runtime entry point, which has type signature **`ompt_get_device_num_procs_t`**, returns the number of processors that are available on the device at the time the routine is called. This value may change between the time that it is determined and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation.

Description of Arguments

The *device* argument is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

Cross References

- **`ompt_device_t`** type, see Section 19.4.4.5.

19.6.2.2 `ompt_get_device_time_t`

Summary

The **`ompt_get_device_time_t`** type is the type signature of the **`ompt_get_device_time`** runtime entry point, which returns the current time on the specified device.

Format

C / C++

```
typedef ompt_device_time_t (*ompt_get_device_time_t) (
  ompt_device_t *device
);
```

C / C++

Semantics

Host and target devices are typically distinct and run independently. If host and target devices are different hardware components, they may use different clock generators. For this reason, a common time base for ordering host-side and device-side events may not be available.

The **ompt_get_device_time** runtime entry point, which has type signature **ompt_get_device_time_t**, returns the current time on the specified device. A tool can use this information to align time stamps from different devices.

Description of Arguments

The *device* argument is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

Cross References

- **ompt_device_t** type, see Section 19.4.4.5.
- **ompt_device_time_t** type, see Section 19.4.4.6.

19.6.2.3 ompt_translate_time_t

Summary

The **ompt_translate_time_t** type is the type signature of the **ompt_translate_time** runtime entry point, which translates a time value that is obtained from the specified device to a corresponding time value on the host device.

Format

C / C++

```
typedef double (*ompt_translate_time_t) (
  ompt_device_t *device,
  ompt_device_time_t time
);
```

C / C++

Semantics

The **ompt_translate_time** runtime entry point, which has type signature **ompt_translate_time_t**, translates a time value obtained from the specified device to a corresponding time value on the host device. The returned value for the host time has the same meaning as the value returned from **omp_get_wtime**.

Note – The accuracy of time translations may degrade, if they are not performed promptly after a device time value is received and if either the host or device vary their clock speeds. Prompt translation of device times to host times is recommended.

Description of Arguments

The *device* argument is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

The *time* argument is a time from the specified device.

Cross References

- **ompt_device_t** type, see Section 19.4.4.5.
- **ompt_device_time_t** type, see Section 19.4.4.6.
- **omp_get_wtime** routine, see Section 18.10.1.

19.6.2.4 ompt_set_trace_ompt_t

Summary

The **ompt_set_trace_ompt_t** type is the type signature of the **ompt_set_trace_ompt** runtime entry point, which enables or disables the recording of trace records for one or more types of OMPT events.

Format

C / C++

```
typedef ompt_set_result_t (*ompt_set_trace_ompt_t) (
  ompt_device_t *device,
  unsigned int enable,
  unsigned int etype
);
```

C / C++

Description of Arguments

The *device* argument points to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

The *etype* argument indicates the events to which the invocation of **ompt_set_trace_ompt** applies. If the value of *etype* is 0 then the invocation applies to all events. If *etype* is positive then it applies to the event in **ompt_callbacks_t** that matches that value.

The *enable* argument indicates whether tracing should be enabled or disabled for the event or events that the *etype* argument specifies. A positive value for *enable* indicates that recording should be enabled; a value of 0 for *enable* indicates that recording should be disabled.

Restrictions

Restrictions on the **ompt_set_trace_ompt** runtime entry point are as follows:

- The entry point must not return **ompt_set_sometimes_paired**.

Cross References

- **ompt_callbacks_t** type, see Section 19.4.2.
- **ompt_device_t** type, see Section 19.4.4.5.
- **ompt_set_result_t** type, see Section 19.4.4.2.
- Tracing activity on target devices with OMPT, see Section 19.2.5.

19.6.2.5 ompt_set_trace_native_t

Summary

The **ompt_set_trace_native_t** type is the type signature of the **ompt_set_trace_native** runtime entry point, which enables or disables the recording of native trace records for a device.

Format

C / C++

```
typedef ompt_set_result_t (*ompt_set_trace_native_t) (
  ompt_device_t *device,
  int enable,
  int flags
);
```

C / C++

Semantics

This interface is designed for use by a tool that cannot directly use native control functions for the device. If a tool can directly use the native control functions then it can invoke native control functions directly using pointers that the *lookup* function associated with the device provides and that are described in the *documentation* string that is provided to the device initializer callback.

Description of Arguments

The *device* argument points to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

The *enable* argument indicates whether this invocation should enable or disable recording of events.

The *flags* argument specifies the kinds of native device monitoring to enable or to disable. Each kind of monitoring is specified by a flag bit. Flags can be composed by using logical `or` to combine enumeration values from type **ompt_native_mon_flag_t**.

To start, to pause, to flush, or to stop tracing for a specific target device associated with *device*, a tool invokes the **ompt_start_trace**, **ompt_pause_trace**, **ompt_flush_trace**, or **ompt_stop_trace** runtime entry point for the device.

Restrictions

Restrictions on the **ompt_set_trace_native** runtime entry point are as follows:

- The entry point must not return **ompt_set_sometimes_paired**.

Cross References

- **ompt_device_t** type, see Section 19.4.4.5.
- **ompt_set_result_t** type, see Section 19.4.4.2.
- Tracing activity on target devices with OMPT, see Section 19.2.5.

19.6.2.6 ompt_start_trace_t

Summary

The **ompt_start_trace_t** type is the type signature of the **ompt_start_trace** runtime entry point, which starts tracing of activity on a specific device.

Format

C / C++

```
typedef int (*ompt_start_trace_t) (
  ompt_device_t *device,
  ompt_callback_buffer_request_t request,
  ompt_callback_buffer_complete_t complete
);
```

C / C++

Semantics

A device's **ompt_start_trace** runtime entry point, which has type signature **ompt_start_trace_t**, initiates tracing on the device. Under normal operating conditions, every event buffer provided to a device by a tool callback is returned to the tool before the OpenMP runtime shuts down. If an exceptional condition terminates execution of an OpenMP program, the OpenMP runtime may not return buffers provided to the device.

An invocation of **ompt_start_trace** returns 1 if the command succeeds and 0 otherwise.

Description of Arguments

The *device* argument points to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

The *request* argument specifies a tool callback that supplies a buffer in which a device can deposit events.

The *complete* argument specifies a tool callback that is invoked by the OpenMP implementation to empty a buffer that contains event records.

Cross References

- **ompt_callback_buffer_complete_t** callback type, see Section 19.5.2.24.
- **ompt_callback_buffer_request_t** callback type, see Section 19.5.2.23.
- **ompt_device_t** type, see Section 19.4.4.5.

19.6.2.7 ompt_pause_trace_t

Summary

The **ompt_pause_trace_t** type is the type signature of the **ompt_pause_trace** runtime entry point, which pauses or restarts activity tracing on a specific device.

Format

C / C++

```
typedef int (*ompt_pause_trace_t) (
  ompt_device_t *device,
  int begin_pause
);
```

C / C++

Semantics

A device's **ompt_pause_trace** runtime entry point, which has type signature **ompt_pause_trace_t**, pauses or resumes tracing on a device. An invocation of **ompt_pause_trace** returns 1 if the command succeeds and 0 otherwise. Redundant pause or resume commands are idempotent and will return the same value as the prior command.

Description of Arguments

The *device* argument points to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

The *begin_pause* argument indicates whether to pause or to resume tracing. To resume tracing, zero should be supplied for *begin_pause*; To pause tracing, any other value should be supplied.

Cross References

- **ompt_device_t** type, see Section 19.4.4.5.

19.6.2.8 ompt_flush_trace_t

Summary

The **ompt_flush_trace_t** type is the type signature of the **ompt_flush_trace** runtime entry point, which causes all pending trace records for the specified device to be delivered.

Format

C / C++

```
typedef int (*ompt_flush_trace_t) (
  ompt_device_t *device
);
```

C / C++

Semantics

A device's **ompt_flush_trace** runtime entry point, which has type signature **ompt_flush_trace_t**, causes the OpenMP implementation to issue a sequence of zero or more buffer completion callbacks to deliver all trace records that have been collected prior to the flush. An invocation of **ompt_flush_trace** returns 1 if the command succeeds and 0 otherwise.

Description of Arguments

The *device* argument points to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

Cross References

- **ompt_device_t** type, see Section 19.4.4.5.

19.6.2.9 ompt_stop_trace_t

Summary

The **ompt_stop_trace_t** type is the type signature of the **ompt_stop_trace** runtime entry point, which stops tracing for a device.

Format

C / C++

```
typedef int (*ompt_stop_trace_t) (
  ompt_device_t *device
);
```

C / C++

Semantics

A device's **ompt_stop_trace** runtime entry point, which has type signature **ompt_stop_trace_t**, halts tracing on the device and requests that any pending trace records are flushed. An invocation of **ompt_stop_trace** returns 1 if the command succeeds and 0 otherwise.

Description of Arguments

The *device* argument points to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

Cross References

- **ompt_device_t** type, see Section 19.4.4.5.

19.6.2.10 ompt_advance_buffer_cursor_t

Summary

The **ompt_advance_buffer_cursor_t** type is the type signature of the **ompt_advance_buffer_cursor** runtime entry point, which advances a trace buffer cursor to the next record.

Format

C / C++

```
typedef int (*ompt_advance_buffer_cursor_t) (
  ompt_device_t *device,
  ompt_buffer_t *buffer,
  size_t size,
  ompt_buffer_cursor_t current,
  ompt_buffer_cursor_t *next
);
```

C / C++

Semantics

A device's **ompt_advance_buffer_cursor** runtime entry point, which has type signature **ompt_advance_buffer_cursor_t**, advances a trace buffer pointer to the next trace record. An invocation of **ompt_advance_buffer_cursor** returns *true* if the advance is successful and the next position in the buffer is valid.

Description of Arguments

The *device* argument points to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

The *buffer* argument indicates a trace buffer that is associated with the cursors.

The argument *size* indicates the size of *buffer* in bytes.

The *current* argument is an opaque buffer cursor.

The *next* argument returns the next value of an opaque buffer cursor.

Cross References

- **ompt_buffer_cursor_t** type, see Section 19.4.4.8.
- **ompt_device_t** type, see Section 19.4.4.5.

19.6.2.11 ompt_get_record_type_t

Summary

The **ompt_get_record_type_t** type is the type signature of the **ompt_get_record_type** runtime entry point, which inspects the type of a trace record.

Format

C / C++

```
typedef ompt_record_t (*ompt_get_record_type_t) (
  ompt_buffer_t *buffer,
  ompt_buffer_cursor_t current
);
```

C / C++

Semantics

Trace records for a device may be in one of two forms: *native* record format, which may be device-specific, or *OMPT* record format, in which each trace record corresponds to an OpenMP *event* and most fields in the record structure are the arguments that would be passed to the OMPT callback for the event.

A device's **ompt_get_record_type** runtime entry point, which has type signature **ompt_get_record_type_t**, inspects the type of a trace record and indicates whether the record at the current position in the trace buffer is an OMPT record, a native record, or an invalid record. An invalid record type is returned if the cursor is out of bounds.

Description of Arguments

The *buffer* argument indicates a trace buffer.

The *current* argument is an opaque buffer cursor.

Cross References

- **ompt_buffer_cursor_t** type, see Section 19.4.4.8.
- **ompt_buffer_t** type, see Section 19.4.4.7.
- **ompt_record_t** type, see Section 19.4.3.1.

19.6.2.12 ompt_get_record_ompt_t

Summary

The **ompt_get_record_ompt_t** type is the type signature of the **ompt_get_record_ompt** runtime entry point, which obtains a pointer to an OMPT trace record from a trace buffer associated with a device.

Format

C / C++

```
typedef ompt_record_ompt_t *(*ompt_get_record_ompt_t) (
  ompt_buffer_t *buffer,
  ompt_buffer_cursor_t current
);
```

C / C++

Semantics

A device's **ompt_get_record_ompt** runtime entry point, which has type signature **ompt_get_record_ompt_t**, returns a pointer that may point to a record in the trace buffer, or it may point to a record in thread local storage in which the information extracted from a record was assembled. The information available for an event depends upon its type.

The return value of the **ompt_record_ompt_t** type includes a field of a union type that can represent information for any OMPT event record type. Another call to the runtime entry point may overwrite the contents of the fields in a record returned by a prior invocation.

Description of Arguments

The *buffer* argument indicates a trace buffer.

The *current* argument is an opaque buffer cursor.

Cross References

- **ompt_buffer_cursor_t** type, see Section 19.4.4.8.
- **ompt_device_t** type, see Section 19.4.4.5.
- **ompt_record_ompt_t** type, see Section 19.4.3.4.

19.6.2.13 ompt_get_record_native_t

Summary

The **ompt_get_record_native_t** type is the type signature of the **ompt_get_record_native** runtime entry point, which obtains a pointer to a native trace record from a trace buffer associated with a device.

Format

C / C++

```
typedef void *(*ompt_get_record_native_t) (
  ompt_buffer_t *buffer,
  ompt_buffer_cursor_t current,
  ompt_id_t *host_op_id
);
```

C / C++

Semantics

A device's **ompt_get_record_native** runtime entry point, which has type signature **ompt_get_record_native_t**, returns a pointer that may point may point into the specified trace buffer, or into thread local storage in which the information extracted from a trace record was assembled. The information available for a native event depends upon its type. If the function returns a non-null result, it will also set the object to which **host_op_id** points to a host-side identifier for the operation that is associated with the record. A subsequent call to **ompt_get_record_native** may overwrite the contents of the fields in a record returned by a prior invocation.

Description of Arguments

The *buffer* argument indicates a trace buffer.

The *current* argument is an opaque buffer cursor.

The *host_op_id* argument is a pointer to an identifier that is returned by the function. The entry point sets the identifier to which *host_op_id* points to the value of a host-side identifier for an operation on a target device that was created when the operation was initiated by the host.

Cross References

- **ompt_buffer_cursor_t** type, see Section 19.4.4.8.
- **ompt_buffer_t** type, see Section 19.4.4.7.
- **ompt_id_t** type, see Section 19.4.4.3.

19.6.2.14 `ompt_get_record_abstract_t`

Summary

The **ompt_get_record_abstract_t** type is the type signature of the **ompt_get_record_abstract** runtime entry point, which summarizes the context of a native (device-specific) trace record.

Format

C / C++

```
typedef ompt_record_abstract_t *(*ompt_get_record_abstract_t) (
  void *native_record
);
```

C / C++

Semantics

An OpenMP implementation may execute on a device that logs trace records in a native (device-specific) format that a tool cannot interpret directly. The **ompt_get_record_abstract** runtime entry point of a device, which has type signature **ompt_get_record_abstract_t**, translates a native trace record into a standard form.

Description of Arguments

The *native_record* argument is a pointer to a native trace record.

Cross References

- **ompt_record_abstract_t** type, see Section 19.4.3.3.

19.6.3 Lookup Entry Points: `ompt_function_lookup_t`

Summary

The **ompt_function_lookup_t** type is the type signature of the lookup runtime entry points that provide pointers to runtime entry points that are part of the OMPT interface.

Format

C / C++

```
typedef void (*ompt_interface_fn_t) (void);

typedef ompt_interface_fn_t (*ompt_function_lookup_t) (
  const char *interface_function_name
);
```

C / C++

Semantics

An OpenMP implementation provides a pointer to a lookup routine that provides pointers to OMPT runtime entry points. When the implementation invokes a tool initializer to configure the OMPT callback interface, it provides a lookup function that provides pointers to runtime entry points that implement routines that are part of the OMPT callback interface. Alternatively, when it invokes a tool initializer to configure the OMPT tracing interface for a device, it provides a lookup function that provides pointers to runtime entry points that implement tracing control routines appropriate for that device.

If the provided function name is unknown to the OpenMP implementation, the function returns *NULL*. In a compliant implementation, the lookup function provided by the tool initializer for the OMPT callback interface returns a valid function pointer for any OMPT runtime entry point name listed in Table 19.1.

A compliant implementation of a lookup function passed to a tool's **`ompt_device_initialize`** callback must provide non-*NULL* function pointers for all strings in Table 19.4, except for **`ompt_set_trace_ompt`** and **`ompt_get_record_ompt`**, as described in Section 19.2.5.

Description of Arguments

The *interface_function_name* argument is a C string that represents the name of a runtime entry point.

Cross References

- Entry points in the OMPT callback interface, see Table 19.1 for a list and Section 19.6.1 for detailed definitions.
- Entry points in the OMPT tracing interface, see Table 19.4 for a list and Section 19.6.2 for detailed definitions.
- Tool initializer for the OMPT callback interface, see Section 19.5.1.1.
- Tool initializer for a device's OMPT tracing interface, see Section 19.2.5.

20 OMPD Interface

This chapter describes OMPD, which is an interface for *third-party tools*. Third-party tools exist in separate processes from the OpenMP program. To provide OMPD support, an OpenMP implementation must provide an OMPD library that the third-party tool can load. An OpenMP implementation does not need to maintain any extra information to support OMPD inquiries from third-party tools unless it is explicitly instructed to do so.

OMPD allows third-party tools such as debuggers to inspect the OpenMP state of a live program or core file in an implementation-agnostic manner. That is, a third-party tool that uses OMPD should work with any conforming OpenMP implementation. An OpenMP implementer provides a library for OMPD that a third-party tool can dynamically load. The third-party tool can use the interface exported by the OMPD library to inspect the OpenMP state of a program. In order to satisfy requests from the third-party tool, the OMPD library may need to read data from the OpenMP program, or to find the addresses of symbols in it. The OMPD library provides this functionality through a callback interface that the third-party tool must instantiate for the OMPD library.

To use OMPD, the third-party tool loads the OMPD library. The OMPD library exports the API that is defined throughout this section, and the third-party tool uses the API to determine OpenMP information about the OpenMP program. The OMPD library must look up the symbols and read data out of the program. It does not perform these operations directly but instead directs the third-party tool to perform them by using the callback interface that the third-party tool exports.

The OMPD design insulates third-party tools from the internal structure of the OpenMP runtime, while the OMPD library is insulated from the details of how to access the OpenMP program. This decoupled design allows for flexibility in how the OpenMP program and third-party tool are deployed, so that, for example, the third-party tool and the OpenMP program are not required to execute on the same machine.

Generally, the third-party tool does not interact directly with the OpenMP runtime but instead interacts with the runtime through the OMPD library. However, a few cases require the third-party tool to access the OpenMP runtime directly. These cases fall into two broad categories. The first is during initialization where the third-party tool must look up symbols and read variables in the OpenMP runtime in order to identify the OMPD library that it should use, which is discussed in Section 20.2.2 and Section 20.2.3. The second category relates to arranging for the third-party tool to be notified when certain events occur during the execution of the OpenMP program. For this purpose, the OpenMP implementation must define certain symbols in the runtime code, as is discussed in Section 20.6. Each of these symbols corresponds to an event type. The OpenMP runtime must ensure that control passes through the appropriate named location when events occur. If the third-party tool requires notification of an event, it can plant a breakpoint at the matching location. The location can, but may not, be a function. It can, for example, simply be a label. However, the names of the locations must have external **C** linkage.

20.1 OMPD Interfaces Definitions

C / C++

A compliant implementation must supply a set of definitions for the OMPD runtime entry points, OMPD third-party tool callback signatures, third-party tool interface functions and the special data types of their parameters and return values. These definitions, which are listed throughout this chapter, and their associated declarations shall be provided in a header file named **omp-tools.h**. In addition, the set of definitions may specify other implementation-specific values.

The **ompd_dll_locations** variable, all OMPD third-party tool interface functions, and all OMPD runtime entry points are external symbols with **C** linkage.

C / C++

20.2 Activating a Third-Party Tool

The third-party tool and the OpenMP program exist as separate processes. Thus, coordination is required between the OpenMP runtime and the third-party tool for OMPD.

20.2.1 Enabling Runtime Support for OMPD

In order to support third-party tools, the OpenMP runtime may need to collect and to store information that it may not otherwise maintain. The OpenMP runtime collects whatever information is necessary to support OMPD if the environment variable **OMP_DEBUG** is set to *enabled*.

Cross References

- **OMP_DEBUG** environment variable, see Section 21.4.1.
- Activating a first-party tool, see Section 19.2.

20.2.2 ompd_dll_locations

Summary

The **ompd_dll_locations** global variable points to the locations of OMPD libraries that are compatible with the OpenMP implementation.

Format

C

```c
extern const char **ompd_dll_locations;
```

C

Semantics

An OpenMP runtime may have more than one OMPD library. The third-party tool must be able to locate the right library to use for the OpenMP program that it is examining. The OpenMP runtime system must provide a public variable **ompd_dll_locations**, which is an **argv**-style vector of filename string pointers that provides the names of any compatible OMPD libraries. This variable must have **C** linkage. The third-party tool uses the name of the variable verbatim and, in particular, does not apply any name mangling before performing the look up.

The architecture on which the third-party tool and, thus, the OMPD library execute does not have to match the architecture on which the OpenMP program that is being examined executes. The third-party tool must interpret the contents of **ompd_dll_locations** to find a suitable OMPD library that matches its own architectural characteristics. On platforms that support different architectures (for example, 32-bit vs 64-bit), OpenMP implementations are encouraged to provide an OMPD library for each supported architecture that can handle OpenMP programs that run on any supported architecture. Thus, for example, a 32-bit debugger that uses OMPD should be able to debug a 64-bit OpenMP program by loading a 32-bit OMPD implementation that can manage a 64-bit OpenMP runtime.

The **ompd_dll_locations** variable points to a *NULL*-terminated vector of zero or more null-terminated pathname strings that do not have any filename conventions. This vector must be fully initialized *before* **ompd_dll_locations** is set to a non-null value. Thus, if a third-party tool, such as a debugger, stops execution of the OpenMP program at any point at which **ompd_dll_locations** is non-null, the vector of strings to which it points shall be valid and complete.

Cross References

- **ompd_dll_locations_valid** global variable, see Section 20.2.3.

20.2.3 ompd_dll_locations_valid

Summary

The OpenMP runtime notifies third-party tools that **ompd_dll_locations** is valid by allowing execution to pass through a location that the symbol **ompd_dll_locations_valid** identifies.

Format

C

```c
void ompd_dll_locations_valid(void);
```

C

Semantics

Since **ompd_dll_locations** may not be a static variable, it may require runtime initialization. The OpenMP runtime notifies third-party tools that **ompd_dll_locations** is valid by having execution pass through a location that the symbol **ompd_dll_locations_valid** identifies. If **ompd_dll_locations** is *NULL*, a third-party tool can place a breakpoint at **ompd_dll_locations_valid** to be notified that **ompd_dll_locations** is initialized. In practice, the symbol **ompd_dll_locations_valid** may not be a function; instead, it may be a labeled machine instruction through which execution passes once the vector is valid.

20.3 OMPD Data Types

This section defines OMPD data types.

20.3.1 Size Type

Summary

The **ompd_size_t** type specifies the number of bytes in opaque data objects that are passed across the OMPD API.

Format

C / C++

```
typedef uint64_t ompd_size_t;
```

C / C++

20.3.2 Wait ID Type

Summary

A variable of **ompd_wait_id_t** type identifies the object on which a thread waits.

Format

C / C++

```
typedef uint64_t ompd_wait_id_t;
```

C / C++

Semantics

The values and meaning of **ompd_wait_id_t** is the same as defined for the **ompt_wait_id_t** type.

Cross References

- **ompt_wait_id_t** type, see Section 19.4.4.31.

20.3.3 Basic Value Types

Summary

These definitions represent word, address, and segment value types.

Format

C / C++

```
typedef uint64_t ompd_addr_t;
typedef int64_t  ompd_word_t;
typedef uint64_t ompd_seg_t;
```

C / C++

Semantics

The *ompd_addr_t* type represents an address in an OpenMP process with an unsigned integer type. The *ompd_word_t* type represents a data word from the OpenMP runtime with a signed integer type. The *ompd_seg_t* type represents a segment value with an unsigned integer type.

20.3.4 Address Type

Summary

The **ompd_address_t** type is used to specify device addresses.

Format

C / C++

```
typedef struct ompd_address_t {
  ompd_seg_t segment;
  ompd_addr_t address;
} ompd_address_t;
```

C / C++

Semantics

The **ompd_address_t** type is a structure that OMPD uses to specify device addresses, which may or may not be segmented. For non-segmented architectures, **ompd_segment_none** is used in the *segment* field of **ompd_address_t**; it is an instance of the **ompd_seg_t** type that has the value 0.

20.3.5 Frame Information Type

Summary

The **ompd_frame_info_t** type is used to specify frame information.

Format

C / C++

```
typedef struct ompd_frame_info_t {
  ompd_address_t frame_address;
  ompd_word_t frame_flag;
} ompd_frame_info_t;
```

C / C++

Semantics

The **ompd_frame_info_t** type is a structure that OMPD uses to specify frame information. The *frame_address* field of **ompd_frame_info_t** identifies a frame. The *frame_flag* field of **ompd_frame_info_t** indicates what type of information is provided in *frame_address*. The values and meaning is the same as defined for the **ompt_frame_flag_t** enumeration type.

Cross References

- **ompt_frame_t** type, see Section 19.4.4.29.

20.3.6 System Device Identifiers

Summary

The **ompd_device_t** type provides information about OpenMP devices.

Format

C / C++

```
typedef uint64_t ompd_device_t;
```

C / C++

Semantics

OpenMP runtimes may utilize different underlying devices, each represented by a device identifier. The device identifiers can vary in size and format and, thus, are not explicitly represented in the OMPD interface. Instead, a device identifier is passed across the interface via its **ompd_device_t** kind, its size in bytes and a pointer to where it is stored. The OMPD library and the third-party tool use the **ompd_device_t** kind to interpret the format of the device identifier that is referenced by the pointer argument. Each different device identifier kind is represented by a unique unsigned 64-bit integer value.

Recommended values of **ompd_device_t** kinds are defined in the **ompd-types.h** header file, which is available on `http://www.openmp.org/`.

20.3.7 Native Thread Identifiers

Summary

The **ompd_thread_id_t** type provides information about native threads.

Format

C / C++

```
typedef uint64_t ompd_thread_id_t;
```

C / C++

Semantics

OpenMP runtimes may use different native thread implementations. Native thread identifiers for these implementations can vary in size and format and, thus, are not explicitly represented in the OMPD interface. Instead, a native thread identifier is passed across the interface via its **ompd_thread_id_t** kind, its size in bytes and a pointer to where it is stored. The OMPD library and the third-party tool use the **ompd_thread_id_t** kind to interpret the format of the native thread identifier that is referenced by the pointer argument. Each different native thread identifier kind is represented by a unique unsigned 64-bit integer value.

Recommended values of **ompd_thread_id_t** kinds, and formats for some corresponding native thread identifiers, are defined in the **ompd-types.h** header file, which is available on http://www.openmp.org/.

20.3.8 OMPD Handle Types

Summary

The OMPD library defines handles for referring to address spaces, threads, parallel regions and tasks that are managed by the OpenMP runtime. The internal structure of the handles are opaque to the third-party tool.

Format

C / C++

```
typedef struct _ompd_aspace_handle ompd_address_space_handle_t;
typedef struct _ompd_thread_handle ompd_thread_handle_t;
typedef struct _ompd_parallel_handle ompd_parallel_handle_t;
typedef struct _ompd_task_handle ompd_task_handle_t;
```

C / C++

Semantics

OMPD uses handles for the following entities that are managed by the OpenMP runtime: address spaces (**ompd_address_space_handle_t**), threads (**ompd_thread_handle_t**), parallel regions (**ompd_parallel_handle_t**), and tasks (**ompd_task_handle_t**). Each operation of the OMPD interface that applies to a particular address space, thread, parallel region or task must explicitly specify a corresponding handle. Handles are defined by the OMPD library and are opaque to the third-party tool. A handle remains constant and valid while the associated entity is managed by the OpenMP runtime or until it is released with the corresponding third-party tool interface routine for releasing handles of that type. If a tool receives notification of the end of the lifetime of a managed entity (see Section 20.6) or it releases the handle, the handle may no longer be referenced.

Defining externally visible type names in this way introduces type safety to the interface, and helps to catch instances where incorrect handles are passed by the third-party tool to the OMPD library. The structures do not need to be defined; instead, the OMPD library must cast incoming (pointers to) handles to the appropriate internal, private types.

20.3.9 OMPD Scope Types

Summary

The **ompd_scope_t** type identifies OMPD scopes.

Format

C / C++

```
typedef enum ompd_scope_t {
  ompd_scope_global         = 1,
  ompd_scope_address_space = 2,
  ompd_scope_thread         = 3,
  ompd_scope_parallel       = 4,
  ompd_scope_implicit_task = 5,
  ompd_scope_task           = 6
} ompd_scope_t;
```

C / C++

Semantics

The **ompd_scope_t** type identifies OpenMP scopes, including those related to parallel regions and tasks. When used in an OMPD interface function call, the scope type and the OMPD handle must match according to Table 20.1.

20.3.10 ICV ID Type

Summary

The **ompd_icv_id_t** type identifies an OpenMP implementation ICV.

TABLE 20.1: Mapping of Scope Type and OMPD Handles

Scope types	Handles
ompd_scope_global	Address space handle for the host device
ompd_scope_address_space	Any address space handle
ompd_scope_thread	Any thread handle
ompd_scope_parallel	Any parallel region handle
ompd_scope_implicit_task	Task handle for an implicit task
ompd_scope_task	Any task handle

Format

C / C++

```
typedef uint64_t ompd_icv_id_t;
```

C / C++

The **ompd_icv_id_t** type identifies OpenMP implementation ICVs. **ompd_icv_undefined** is an instance of this type with the value 0.

20.3.11 Tool Context Types

Summary

A third-party tool defines contexts to identify abstractions uniquely. The internal structure of these contexts are opaque to the OMPD library.

Format

C / C++

```
typedef struct _ompd_aspace_cont ompd_address_space_context_t;
typedef struct _ompd_thread_cont ompd_thread_context_t;
```

C / C++

Semantics

A third-party tool uniquely defines an *address space context* to identify the address space for the process that it is monitoring. Similarly, it uniquely defines a *thread context* to identify a native thread of the process that it is monitoring. These contexts are opaque to the OMPD library.

20.3.12 Return Code Types

Summary

The **ompd_rc_t** type is the return code type of an OMPD operation.

Format

C / C++

```
typedef enum ompd_rc_t {
  ompd_rc_ok                    = 0,
  ompd_rc_unavailable           = 1,
  ompd_rc_stale_handle          = 2,
  ompd_rc_bad_input             = 3,
  ompd_rc_error                 = 4,
  ompd_rc_unsupported           = 5,
  ompd_rc_needs_state_tracking = 6,
  ompd_rc_incompatible          = 7,
  ompd_rc_device_read_error     = 8,
  ompd_rc_device_write_error    = 9,
  ompd_rc_nomem                 = 10,
  ompd_rc_incomplete            = 11,
  ompd_rc_callback_error        = 12
} ompd_rc_t;
```

C / C++

Semantics

The **ompd_rc_t** type is used for the return codes of OMPD operations. The return code types and their semantics are defined as follows:

- **ompd_rc_ok** is returned when the operation is successful;
- **ompd_rc_unavailable** is returned when information is not available for the specified context;
- **ompd_rc_stale_handle** is returned when the specified handle is no longer valid;
- **ompd_rc_bad_input** is returned when the input parameters (other than handle) are invalid;
- **ompd_rc_error** is returned when a fatal error occurred;
- **ompd_rc_unsupported** is returned when the requested operation is not supported;
- **ompd_rc_needs_state_tracking** is returned when the state tracking operation failed because state tracking is not currently enabled;
- **ompd_rc_device_read_error** is returned when a read operation failed on the device;
- **ompd_rc_device_write_error** is returned when a write operation failed on the device;
- **ompd_rc_incompatible** is returned when this OMPD library is incompatible with the OpenMP program or is not capable of handling it;
- **ompd_rc_nomem** is returned when a memory allocation fails;

- **ompd_rc_incomplete** is returned when the information provided on return is incomplete, while the arguments are still set to valid values; and
- **ompd_rc_callback_error** is returned when the callback interface or any one of the required callback routines provided by the third-party tool is invalid.

20.3.13 Primitive Type Sizes

Summary

The **ompd_device_type_sizes_t** type provides the size of primitive types in the OpenMP architecture address space.

Format

C / C++

```
typedef struct ompd_device_type_sizes_t {
  uint8_t sizeof_char;
  uint8_t sizeof_short;
  uint8_t sizeof_int;
  uint8_t sizeof_long;
  uint8_t sizeof_long_long;
  uint8_t sizeof_pointer;
} ompd_device_type_sizes_t;
```

C / C++

Semantics

The **ompd_device_type_sizes_t** type is used in operations through which the OMPD library can interrogate the third-party tool about the size of primitive types for the target architecture of the OpenMP runtime, as returned by the **sizeof** operator. The fields of **ompd_device_type_sizes_t** give the sizes of the eponymous basic types used by the OpenMP runtime. As the third-party tool and the OMPD library, by definition, execute on the same architecture, the size of the fields can be given as **uint8_t**.

Cross References

- **ompd_callback_sizeof_fn_t** type, see Section 20.4.2.2.

20.4 OMPD Third-Party Tool Callback Interface

For the OMPD library to provide information about the internal state of the OpenMP runtime system in an OpenMP process or core file, it must have a means to extract information from the OpenMP process that the third-party tool is examining. The OpenMP process on which the third-party tool is operating may be either a “live” process or a core file, and a thread may be either a “live” thread in an OpenMP process or a thread in a core file. To enable the OMPD library to extract state information from an OpenMP process or core file, the third-party tool must supply the OMPD library with callback functions to inquire about the size of primitive types in the device of the OpenMP process, to look up the addresses of symbols, and to read and to write memory in the device. The OMPD library uses these callbacks to implement its interface operations. The OMPD library only invokes the callback functions in direct response to calls made by the third-party tool to the OMPD library.

Description of Return Codes

All of the OMPD callback functions must return the following return codes or function-specific return codes:

- **ompd_rc_ok** on success; or
- **ompd_rc_stale_handle** if an invalid context argument is provided.

20.4.1 Memory Management of OMPD Library

ompd_callback_memory_alloc_fn_t (see Section 20.4.1.1) and **ompd_callback_memory_free_fn_t** (see Section 20.4.1.2), which are provided by the third-party tool to obtain and to release heap memory. This mechanism ensures that the library does not interfere with any custom memory management scheme that the third-party tool may use.

If the OMPD library is implemented in C++ then memory management operators, like **new** and **delete** and their variants, *must all* be overloaded and implemented in terms of the callbacks that the third-party tool provides. The OMPD library must be implemented in a manner such that any of its definitions of **new** or **delete** do not interfere with any that the third-party tool defines.

In some cases, the OMPD library must allocate memory to return results to the third-party tool. The third-party tool then owns this memory and has the responsibility to release it. Thus, the OMPD library and the third-party tool must use the same memory manager.

The OMPD library creates OMPD handles, which are opaque to the third-party tool and may have a complex internal structure. The third-party tool cannot determine if the handle pointers that the API returns correspond to discrete heap allocations. Thus, the third-party tool must not simply deallocate a handle by passing an address that it receives from the OMPD library to its own memory manager. Instead, the OMPD API includes functions that the third-party tool must use when it no longer needs a handle.

A third-party tool creates contexts and passes them to the OMPD library. The OMPD library does not release contexts; instead the third-party tool releases them after it releases any handles that may reference the contexts.

20.4.1.1 `ompd_callback_memory_alloc_fn_t`

Summary

The **`ompd_callback_memory_alloc_fn_t`** type is the type signature of the callback routine that the third-party tool provides to the OMPD library to allocate memory.

Format

C

```c
typedef ompd_rc_t (*ompd_callback_memory_alloc_fn_t) (
  ompd_size_t nbytes,
  void **ptr
);
```

C

Semantics

The **`ompd_callback_memory_alloc_fn_t`** type is the type signature of the memory allocation callback routine that the third-party tool provides. The OMPD library may call the **`ompd_callback_memory_alloc_fn_t`** callback function to allocate memory.

Description of Arguments

The *nbytes* argument is the size in bytes of the block of memory to allocate.

The address of the newly allocated block of memory is returned in the location to which the *ptr* argument points. The newly allocated block is suitably aligned for any type of variable and is not guaranteed to be set to zero.

Description of Return Codes

Routines that use the **`ompd_callback_memory_alloc_fn_t`** type may return the general return codes listed at the beginning of Section 20.4.

Cross References

- **`ompd_rc_t`** type, see Section 20.3.12.
- **`ompd_size_t`** type, see Section 20.3.1.

20.4.1.2 `ompd_callback_memory_free_fn_t`

Summary

The **`ompd_callback_memory_free_fn_t`** type is the type signature of the callback routine that the third-party tool provides to the OMPD library to deallocate memory.

Format

C

```
typedef ompd_rc_t (*ompd_callback_memory_free_fn_t) (
  void *ptr
);
```

C

Semantics

The **ompd_callback_memory_free_fn_t** type is the type signature of the memory deallocation callback routine that the third-party tool provides. The OMPD library may call the **ompd_callback_memory_free_fn_t** callback function to deallocate memory that was obtained from a prior call to the **ompd_callback_memory_alloc_fn_t** callback function.

Description of Arguments

The *ptr* argument is the address of the block to be deallocated.

Description of Return Codes

Routines that use the **ompd_callback_memory_free_fn_t** type may return the general return codes listed at the beginning of Section 20.4.

Cross References

- **ompd_callbacks_t** type, see Section 20.4.6.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_callback_memory_alloc_fn_t** type, see Section 20.4.1.1.

20.4.2 Context Management and Navigation

Summary

The third-party tool provides the OMPD library with callbacks to manage and to navigate context relationships.

20.4.2.1 ompd_callback_get_thread_context_for_thread_id_fn_t

Summary

The **ompd_callback_get_thread_context_for_thread_id_fn_t** is the type signature of the callback routine that the third-party tool provides to the OMPD library to map a native thread identifier to a third-party tool thread context.

Format

C

```c
typedef ompd_rc_t
(*ompd_callback_get_thread_context_for_thread_id_fn_t) (
  ompd_address_space_context_t *address_space_context,
  ompd_thread_id_t kind,
  ompd_size_t sizeof_thread_id,
  const void *thread_id,
  ompd_thread_context_t **thread_context
);
```

C

Semantics

The **ompd_callback_get_thread_context_for_thread_id_fn_t** is the type signature of the context mapping callback routine that the third-party tool provides. This callback maps a native thread identifier to a third-party tool thread context. The native thread identifier is within the address space that *address_space_context* identifies. The OMPD library can use the thread context, for example, to access thread local storage.

Description of Arguments

The *address_space_context* argument is an opaque handle that the third-party tool provides to reference an address space. The *kind*, *sizeof_thread_id*, and *thread_id* arguments represent a native thread identifier. On return, the *thread_context* argument provides an opaque handle that maps a native thread identifier to a third-party tool thread context.

Description of Return Codes

In addition to the general return codes listed at the beginning of Section 20.4, routines that use the **ompd_callback_get_thread_context_for_thread_id_fn_t** type may also return the following return codes:

- **ompd_rc_bad_input** if a different value in *sizeof_thread_id* is expected for the native thread identifier kind given by *kind*; or
- **ompd_rc_unsupported** if the native thread identifier *kind* is not supported.

Restrictions

Restrictions on routines that use **ompd_callback_get_thread_context_for_thread_id_fn_t** are as follows:

- The provided *thread_context* must be valid until the OMPD library returns from the OMPD third-party tool interface routine.

Cross References

- **ompd_thread_id_t** type, see Section 20.3.7.
- **ompd_address_space_context_t** type, see Section 20.3.11.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_thread_context_t** type, see Section 20.3.11.
- **ompd_size_t** type, see Section 20.3.1.

20.4.2.2 ompd_callback_sizeof_fn_t

Summary

The **ompd_callback_sizeof_fn_t** type is the type signature of the callback routine that the third-party tool provides to the OMPD library to determine the sizes of the primitive types in an address space.

Format

C

```
typedef ompd_rc_t (*ompd_callback_sizeof_fn_t) (
  ompd_address_space_context_t *address_space_context,
  ompd_device_type_sizes_t *sizes
);
```

C

Semantics

The **ompd_callback_sizeof_fn_t** is the type signature of the type-size query callback routine that the third-party tool provides. This callback provides the sizes of the basic primitive types for a given address space.

Description of Arguments

The callback returns the sizes of the basic primitive types used by the address space context that the *address_space_context* argument specifies in the location to which the *sizes* argument points.

Description of Return Codes

Routines that use the **ompd_callback_sizeof_fn_t** type may return the general return codes listed at the beginning of Section 20.4.

Cross References

- **ompd_address_space_context_t** type, see Section 20.3.11.
- **ompd_callbacks_t** type, see Section 20.4.6.
- **ompd_device_type_sizes_t** type, see Section 20.3.13.
- **ompd_rc_t** type, see Section 20.3.12.

20.4.3 Accessing Memory in the OpenMP Program or Runtime

20.4.3.1 `ompd_callback_symbol_addr_fn_t`

Summary

The **`ompd_callback_symbol_addr_fn_t`** type is the type signature of the callback that the third-party tool provides to look up the addresses of symbols in an OpenMP program.

Format

C

```
typedef ompd_rc_t (*ompd_callback_symbol_addr_fn_t) (
  ompd_address_space_context_t *address_space_context,
  ompd_thread_context_t *thread_context,
  const char *symbol_name,
  ompd_address_t *symbol_addr,
  const char *file_name
);
```

C

Semantics

The **`ompd_callback_symbol_addr_fn_t`** is the type signature of the symbol-address query callback routine that the third-party tool provides. This callback looks up addresses of symbols within a specified address space.

Description of Arguments

This callback looks up the symbol provided in the *symbol_name* argument.

The *address_space_context* argument is the third-party tool's representation of the address space of the process, core file, or device.

The *thread_context* argument is *NULL* for global memory accesses. If *thread_context* is not *NULL*, *thread_context* gives the thread-specific context for the symbol lookup for the purpose of calculating thread local storage addresses. In this case, the thread to which *thread_context* refers must be associated with either the process or the device that corresponds to the *address_space_context* argument.

The third-party tool uses the *symbol_name* argument that the OMPD library supplies verbatim. In particular, no name mangling, demangling or other transformations are performed prior to the lookup. The *symbol_name* parameter must correspond to a statically allocated symbol within the specified address space. The symbol can correspond to any type of object, such as a variable, thread local storage variable, function, or untyped label. The symbol can have a local, global, or weak binding.

The *file_name* argument is an optional input parameter that indicates the name of the shared library in which the symbol is defined, and it is intended to help the third-party tool disambiguate symbols that are defined multiple times across the executable or shared library files. The shared library name may not be an exact match for the name seen by the third-party tool. If *file_name* is *NULL* then the third-party tool first tries to find the symbol in the executable file, and, if the symbol is not found, the third-party tool tries to find the symbol in the shared libraries in the order in which the shared libraries are loaded into the address space. If *file_name* is non-null then the third-party tool first tries to find the symbol in the libraries that match the name in the *file_name* argument, and, if the symbol is not found, the third-party tool then uses the same procedure as when *file_name* is *NULL*.

The callback does not support finding either symbols that are dynamically allocated on the call stack or statically allocated symbols that are defined within the scope of a function or subroutine.

The callback returns the address of the symbol in the location to which *symbol_addr* points.

Description of Return Codes

In addition to the general return codes listed at the beginning of Section 20.4, routines that use the **`ompd_callback_symbol_addr_fn_t`** type may also return the following return codes:

- **`ompd_rc_error`** if the requested symbol is not found; or
- **`ompd_rc_bad_input`** if no symbol name is provided.

Restrictions

Restrictions on routines that use the **`ompd_callback_symbol_addr_fn_t`** type are as follows:

- The *address_space_context* argument must be non-null.
- The symbol that the *symbol_name* argument specifies must be defined.

Cross References

- **`ompd_address_space_context_t`** type, see Section 20.3.11.
- **`ompd_callbacks_t`** type, see Section 20.4.6.
- **`ompd_rc_t`** type, see Section 20.3.12.
- **`ompd_thread_context_t`** type, see Section 20.3.11.
- **`ompd_address_t`** type, see Section 20.3.4.

20.4.3.2 `ompd_callback_memory_read_fn_t`

Summary

The **`ompd_callback_memory_read_fn_t`** type is the type signature of the callback that the third-party tool provides to read data (*read_memory*) or a string (*read_string*) from an OpenMP program.

Format

C

```c
typedef ompd_rc_t (*ompd_callback_memory_read_fn_t) (
  ompd_address_space_context_t *address_space_context,
  ompd_thread_context_t *thread_context,
  const ompd_address_t *addr,
  ompd_size_t nbytes,
  void *buffer
);
```

C

Semantics

The **ompd_callback_memory_read_fn_t** is the type signature of the read callback routines that the third-party tool provides.

The *read_memory* callback copies a block of data from *addr* within the address space given by *address_space_context* to the third-party tool *buffer*.

The *read_string* callback copies a string to which *addr* points, including the terminating null byte (**'\0'**), to the third-party tool *buffer*. At most *nbytes* bytes are copied. If a null byte is not among the first *nbytes* bytes, the string placed in *buffer* is not null-terminated.

Description of Arguments

The address from which the data are to be read in the OpenMP program that *address_space_context* specifies is given by *addr*. The *nbytes* argument is the number of bytes to be transferred. The *thread_context* argument is optional for global memory access, and in that case should be *NULL*. If it is non-null, *thread_context* identifies the thread-specific context for the memory access for the purpose of accessing thread local storage.

The data are returned through *buffer*, which is allocated and owned by the OMPD library. The contents of the buffer are unstructured, raw bytes. The OMPD library must arrange for any transformations such as byte-swapping that may be necessary (see Section 20.4.4) to interpret the data.

Description of Return Codes

In addition to the general return codes listed at the beginning of Section 20.4, routines that use the **ompd_callback_memory_read_fn_t** type may also return the following return codes:

- **ompd_rc_incomplete** if no terminating null byte is found while reading *nbytes* using the *read_string* callback; or
- **ompd_rc_error** if unallocated memory is reached while reading *nbytes* using either the *read_memory* or *read_string* callback.

Cross References

- **ompd_address_space_context_t** type, see Section 20.3.11.
- **ompd_callbacks_t** type, see Section 20.4.6.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_thread_context_t** type, see Section 20.3.11.
- **ompd_address_t** type, see Section 20.3.4.
- **ompd_callback_device_host_fn_t** type, see Section 20.4.4.
- **ompd_size_t** type, see Section 20.3.1.

20.4.3.3 ompd_callback_memory_write_fn_t

Summary

The **ompd_callback_memory_write_fn_t** type is the type signature of the callback that the third-party tool provides to write data to an OpenMP program.

Format

C

```
typedef ompd_rc_t (*ompd_callback_memory_write_fn_t) (
  ompd_address_space_context_t *address_space_context,
  ompd_thread_context_t *thread_context,
  const ompd_address_t *addr,
  ompd_size_t nbytes,
  const void *buffer
);
```

C

Semantics

The **ompd_callback_memory_write_fn_t** is the type signature of the write callback routine that the third-party tool provides. The OMPD library may call this callback to have the third-party tool write a block of data to a location within an address space from a provided buffer.

Description of Arguments

The address to which the data are to be written in the OpenMP program that *address_space_context* specifies is given by *addr*. The *nbytes* argument is the number of bytes to be transferred. The *thread_context* argument is optional for global memory access, and in that case should be *NULL*. If it is non-null then *thread_context* identifies the thread-specific context for the memory access for the purpose of accessing thread local storage.

The data to be written are passed through *buffer*, which is allocated and owned by the OMPD library. The contents of the buffer are unstructured, raw bytes. The OMPD library must arrange for any transformations such as byte-swapping that may be necessary (see Section 20.4.4) to render the data into a form that is compatible with the OpenMP runtime.

Description of Return Codes

Routines that use the **ompd_callback_memory_write_fn_t** type may return the general return codes listed at the beginning of Section 20.4.

Cross References

- **ompd_address_space_context_t** type, see Section 20.3.11.
- **ompd_callbacks_t** type, see Section 20.4.6.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_thread_context_t** type, see Section 20.3.11.
- **ompd_address_t** type, see Section 20.3.4.
- **ompd_callback_device_host_fn_t** type, see Section 20.4.4.
- **ompd_size_t** type, see Section 20.3.1.

20.4.4 Data Format Conversion: ompd_callback_device_host_fn_t

Summary

The **ompd_callback_device_host_fn_t** type is the type signature of the callback that the third-party tool provides to convert data between the formats that the third-party tool and the OMPD library use and that the OpenMP program uses.

Format

C

```
typedef ompd_rc_t (*ompd_callback_device_host_fn_t) (
  ompd_address_space_context_t *address_space_context,
  const void *input,
  ompd_size_t unit_size,
  ompd_size_t count,
  void *output
);
```

C

Semantics

The architecture on which the third-party tool and the OMPD library execute may be different from the architecture on which the OpenMP program that is being examined executes. Thus, the conventions for representing data may differ. The callback interface includes operations to convert between the conventions, such as the byte order (endianness), that the third-party tool and OMPD library use and the ones that the OpenMP program use. The callback with the **ompd_callback_device_host_fn_t** type signature converts data between the formats.

Description of Arguments

The *address_space_context* argument specifies the OpenMP address space that is associated with the data. The *input* argument is the source buffer and the *output* argument is the destination buffer. The *unit_size* argument is the size of each of the elements to be converted. The *count* argument is the number of elements to be transformed.

The OMPD library allocates and owns the input and output buffers. It must ensure that the buffers have the correct size and are eventually deallocated when they are no longer needed.

Description of Return Codes

Routines that use the **ompd_callback_device_host_fn_t** type may return the general return codes listed at the beginning of Section 20.4.

Cross References

- **ompd_address_space_context_t** type, see Section 20.3.11.
- **ompd_callbacks_t** type, see Section 20.4.6.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_size_t** type, see Section 20.3.1.

20.4.5 ompd_callback_print_string_fn_t

Summary

The **ompd_callback_print_string_fn_t** type is the type signature of the callback that the third-party tool provides so that the OMPD library can emit output.

Format

C

```
typedef ompd_rc_t (*ompd_callback_print_string_fn_t) (
  const char *string,
  int category
);
```

C

Semantics

The OMPD library may call the **ompd_callback_print_string_fn_t** callback function to emit output, such as logging or debug information. The third-party tool may set the **ompd_callback_print_string_fn_t** callback function to *NULL* to prevent the OMPD library from emitting output. The OMPD library may not write to file descriptors that it did not open.

Description of Arguments

The *string* argument is the null-terminated string to be printed. No conversion or formatting is performed on the string.

The *category* argument is the implementation-defined category of the string to be printed.

Description of Return Codes

Routines that use the **ompd_callback_print_string_fn_t** type may return the general return codes listed at the beginning of Section 20.4.

Cross References

- **ompd_callbacks_t** type, see Section 20.4.6.
- **ompd_rc_t** type, see Section 20.3.12.

20.4.6 The Callback Interface

Summary

All OMPD library interactions with the OpenMP program must be through a set of callbacks that the third-party tool provides. These callbacks must also be used for allocating or releasing resources, such as memory, that the OMPD library needs.

Format

C

```
typedef struct ompd_callbacks_t {
  ompd_callback_memory_alloc_fn_t alloc_memory;
  ompd_callback_memory_free_fn_t free_memory;
  ompd_callback_print_string_fn_t print_string;
  ompd_callback_sizeof_fn_t sizeof_type;
  ompd_callback_symbol_addr_fn_t symbol_addr_lookup;
  ompd_callback_memory_read_fn_t read_memory;
  ompd_callback_memory_write_fn_t write_memory;
  ompd_callback_memory_read_fn_t read_string;
  ompd_callback_device_host_fn_t device_to_host;
  ompd_callback_device_host_fn_t host_to_device;
  ompd_callback_get_thread_context_for_thread_id_fn_t
    get_thread_context_for_thread_id;
} ompd_callbacks_t;
```

C

Semantics

The set of callbacks that the OMPD library must use is collected in the **ompd_callbacks_t** structure. An instance of this type is passed to the OMPD library as a parameter to **ompd_initialize** (see Section 20.5.1.1). Each field points to a function that the OMPD library must use either to interact with the OpenMP program or for memory operations.

The *alloc_memory* and *free_memory* fields are pointers to functions the OMPD library uses to allocate and to release dynamic memory.

The *print_string* field points to a function that prints a string.

The architecture on which the OMPD library and third-party tool execute may be different from the architecture on which the OpenMP program that is being examined executes. The *sizeof_type* field points to a function that allows the OMPD library to determine the sizes of the basic integer and pointer types that the OpenMP program uses. Because of the potential differences in the targeted architectures, the conventions for representing data in the OMPD library and the OpenMP program may be different. The *device_to_host* field points to a function that translates data from the conventions that the OpenMP program uses to those that the third-party tool and OMPD library use. The reverse operation is performed by the function to which the *host_to_device* field points.

The *symbol_addr_lookup* field points to a callback that the OMPD library can use to find the address of a global or thread local storage symbol. The *read_memory*, *read_string* and *write_memory* fields are pointers to functions for reading from and writing to global memory or thread local storage in the OpenMP program.

The *get_thread_context_for_thread_id* field is a pointer to a function that the OMPD library can use to obtain a thread context that corresponds to a native thread identifier.

Cross References

- **ompd_callback_device_host_fn_t** type, see Section 20.4.4.
- **ompd_callback_get_thread_context_for_thread_id_fn_t** type, see Section 20.4.2.1.
- **ompd_callback_memory_alloc_fn_t** type, see Section 20.4.1.1.
- **ompd_callback_memory_free_fn_t** type, see Section 20.4.1.2.
- **ompd_callback_memory_read_fn_t** type, see Section 20.4.3.2.
- **ompd_callback_memory_write_fn_t** type, see Section 20.4.3.3.
- **ompd_callback_print_string_fn_t** type, see Section 20.4.5
- **ompd_callback_sizeof_fn_t** type, see Section 20.4.2.2.
- **ompd_callback_symbol_addr_fn_t** type, see Section 20.4.3.1.

20.5 OMPD Tool Interface Routines

This section defines the interface provided by the OMPD library to be used by the third-party tool. Some interface routines require one or more specified threads to be *stopped* for the returned values to be meaningful. In this context, a stopped thread is a thread that is not modifying the observable OpenMP runtime state.

Description of Return Codes

All of the OMPD Tool Interface Routines must return function specific return codes or any of the following return codes:

- **ompd_rc_stale_handle** if a provided handle is stale;
- **ompd_rc_bad_input** if *NULL* is provided for any input argument unless otherwise specified;
- **ompd_rc_callback** if a callback returned an unexpected error, which leads to a failure of the query;
- **ompd_rc_needs_state_tracking** if the information cannot be provided while the *debug-var* is disabled;
- **ompd_rc_ok** on success; or
- **ompd_rc_error** for any other error.

20.5.1 Per OMPD Library Initialization and Finalization

ompd_get_api_version (see Section 20.5.1.2). If the tool supports the version that **ompd_get_api_version** returns, the tool starts the initialization by calling **ompd_initialize** (see Section 20.5.1.1) using the version of the OMPD API that the library supports. If the tool does not support the version that **ompd_get_api_version** returns, it may attempt to call **ompd_initialize** with a different version.

20.5.1.1 ompd_initialize

Summary

The **ompd_initialize** function initializes the OMPD library.

Format

C

```c
ompd_rc_t ompd_initialize(
  ompd_word_t api_version,
  const ompd_callbacks_t *callbacks
);
```

C

Semantics

A tool that uses OMPD calls **ompd_initialize** to initialize each OMPD library that it loads. More than one library may be present in a third-party tool, such as a debugger, because the tool may control multiple devices, which may use different runtime systems that require different OMPD libraries. This initialization must be performed exactly once before the tool can begin to operate on an OpenMP process or core file.

Description of Arguments

The *api_version* argument is the OMPD API version that the tool requests to use. The tool may call **ompd_get_api_version** to obtain the latest OMPD API version that the OMPD library supports.

The tool provides the OMPD library with a set of callback functions in the *callbacks* input argument which enables the OMPD library to allocate and to deallocate memory in the tool's address space, to lookup the sizes of basic primitive types in the device, to lookup symbols in the device, and to read and to write memory in the device.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or any of the following return codes:

- **ompd_rc_bad_input** if invalid callbacks are provided; or
- **ompd_rc_unsupported** if the requested API version cannot be provided.

Cross References

- **ompd_callbacks_t** type, see Section 20.4.6.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_get_api_version** routine, see Section 20.5.1.2.

20.5.1.2 ompd_get_api_version

Summary

The **ompd_get_api_version** function returns the OMPD API version.

Format

C

```c
ompd_rc_t ompd_get_api_version(ompd_word_t *version);
```

C

Semantics

The tool may call the **ompd_get_api_version** function to obtain the latest OMPD API version number of the OMPD library. The OMPD API version number is equal to the value of the **_OPENMP** macro defined in the associated OpenMP implementation, if the C preprocessor is supported. If the associated OpenMP implementation compiles Fortran codes without the use of a C preprocessor, the OMPD API version number is equal to the value of the Fortran integer parameter **openmp_version**.

Description of Arguments

The latest version number is returned into the location to which the *version* argument points.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_rc_t** type, see Section 20.3.12.

20.5.1.3 ompd_get_version_string

Summary

The **ompd_get_version_string** function returns a descriptive string for the OMPD library version.

Format

C

```
ompd_rc_t ompd_get_version_string(const char **string);
```

C

Semantics

The tool may call this function to obtain a pointer to a descriptive version string of the OMPD library vendor, implementation, internal version, date, or any other information that may be useful to a tool user or vendor. An implementation should provide a different string for every change to its source code or build that could be visible to the interface user.

Description of Arguments

A pointer to a descriptive version string is placed into the location to which the *string* output argument points. The OMPD library owns the string that the OMPD library returns; the tool must not modify or release this string. The string remains valid for as long as the library is loaded. The **ompd_get_version_string** function may be called before **ompd_initialize** (see Section 20.5.1.1). Accordingly, the OMPD library must not use heap or stack memory for the string.

The signatures of **ompd_get_api_version** (see Section 20.5.1.2) and **ompd_get_version_string** are guaranteed not to change in future versions of the API. In contrast, the type definitions and prototypes in the rest of the API do not carry the same guarantee. Therefore a tool that uses OMPD should check the version of the API of the loaded OMPD library before it calls any other function of the API.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_rc_t** type, see Section 20.3.12.

20.5.1.4 ompd_finalize

Summary

When the tool is finished with the OMPD library it should call **ompd_finalize** before it unloads the library.

Format

C

```c
ompd_rc_t ompd_finalize(void);
```

C

Semantics

The call to **ompd_finalize** must be the last OMPD call that the tool makes before it unloads the library. This call allows the OMPD library to free any resources that it may be holding.

The OMPD library may implement a *finalizer* section, which executes as the library is unloaded and therefore after the call to **ompd_finalize**. During finalization, the OMPD library may use the callbacks that the tool provided earlier during the call to **ompd_initialize**.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_unsupported** if the OMPD library is not initialized.

Cross References

- **ompd_rc_t** type, see Section 20.3.12.

20.5.2 Per OpenMP Process Initialization and Finalization

20.5.2.1 ompd_process_initialize

Summary

A tool calls **ompd_process_initialize** to obtain an address space handle for the host device when it initializes a session on a live process or core file.

Format

C

```
ompd_rc_t ompd_process_initialize(
  ompd_address_space_context_t *context,
  ompd_address_space_handle_t **host_handle
);
```

C

Semantics

A tool calls **ompd_process_initialize** to obtain an address space handle for the host device when it initializes a session on a live process or core file. On return from **ompd_process_initialize**, the tool owns the address space handle, which it must release with **ompd_rel_address_space_handle**. The initialization function must be called before any OMPD operations are performed on the OpenMP process or core file. This call allows the OMPD library to confirm that it can handle the OpenMP process or core file that *context* identifies.

Description of Arguments

The *context* argument is an opaque handle that the tool provides to address an address space from the host device. On return, the *host_handle* argument provides an opaque handle to the tool for this address space, which the tool must release when it is no longer needed.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_incompatible** if the OMPD library is incompatible with the runtime library loaded in the process.

Cross References

- **ompd_address_space_context_t** type, see Section 20.3.11.
- **ompd_address_space_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_rel_address_space_handle** routine, see Section 20.5.2.3.

20.5.2.2 ompd_device_initialize

Summary

A tool calls **ompd_device_initialize** to obtain an address space handle for a non-host device that has at least one active target region.

Format

C

```
ompd_rc_t ompd_device_initialize(
  ompd_address_space_handle_t *host_handle,
  ompd_address_space_context_t *device_context,
  ompd_device_t kind,
  ompd_size_t sizeof_id,
  void *id,
  ompd_address_space_handle_t **device_handle
);
```

C

Semantics

A tool calls **ompd_device_initialize** to obtain an address space handle for a non-host device that has at least one active target region. On return from **ompd_device_initialize**, the tool owns the address space handle.

Description of Arguments

The *host_handle* argument is an opaque handle that the tool provides to reference the host device address space associated with an OpenMP process or core file. The *device_context* argument is an opaque handle that the tool provides to reference a non-host device address space. The *kind*, *sizeof_id*, and *id* arguments represent a device identifier. On return the *device_handle* argument provides an opaque handle to the tool for this address space.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_unsupported** if the OMPD library has no support for the specific device.

Cross References

- **ompd_device_t** type, see Section 20.3.6.
- **ompd_address_space_context_t** type, see Section 20.3.11.
- **ompd_address_space_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_size_t** type, see Section 20.3.1.

20.5.2.3 ompd_rel_address_space_handle

Summary

A tool calls **ompd_rel_address_space_handle** to release an address space handle.

Format

C

```c
ompd_rc_t ompd_rel_address_space_handle(
  ompd_address_space_handle_t *handle
);
```

C

Semantics

When the tool is finished with the OpenMP process address space handle it should call **ompd_rel_address_space_handle** to release the handle, which allows the OMPD library to release any resources that it has related to the address space.

Description of Arguments

The *handle* argument is an opaque handle for the address space to be released.

Restrictions

Restrictions to the **ompd_rel_address_space_handle** routine are as follows:

- An address space context must not be used after the corresponding address space handle is released.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_address_space_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.

20.5.2.4 `ompd_get_device_thread_id_kinds`

Summary

The **`ompd_get_device_thread_id_kinds`** function returns a list of supported native thread identifer kinds and a corresponding list of their respective sizes.

Format

C

```c
ompd_rc_t ompd_get_device_thread_id_kinds(
  ompd_address_space_handle_t *device_handle,
  ompd_thread_id_t *kinds,
  ompd_size_t *thread_id_sizes,
  int *count
);
```

C

Semantics

The **`ompd_get_device_thread_id_kinds`** function returns an array of supported native thread identifer kinds and a corresponding array of their respective sizes for a given device. The OMPD library allocates storage for the arrays with the memory allocation callback that the tool provides. Each supported native thread identifier kind is guaranteed to be recognizable by the OMPD library and may be mapped to and from any OpenMP thread that executes on the device.

The third-party tool owns the storage for the array of kinds and the array of sizes that is returned via the *kinds* and *thread_id_sizes* arguments, and it is responsible for freeing that storage.

Description of Arguments

The *device_handle* argument is a pointer to an opaque address space handle that represents a host device (returned by **`ompd_process_initialize`**) or a non-host device (returned by **`ompd_device_initialize`**). On return, the *kinds* argument is the address of an array of native thread identifier kinds, the *thread_id_sizes* argument is the address of an array of the corresponding native thread identifier sizes used by the OMPD library, and the *count* argument is the address of a variable that indicates the sizes of the returned arrays.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **`ompd_thread_id_t`** type, see Section 20.3.7.
- **`ompd_address_space_handle_t`** type, see Section 20.3.8.
- **`ompd_rc_t`** type, see Section 20.3.12.
- **`ompd_size_t`** type, see Section 20.3.1.

20.5.3 Thread and Signal Safety

The OMPD library does not need to be reentrant. The tool must ensure that only one thread enters the OMPD library at a time. The OMPD library must not install signal handlers or otherwise interfere with the tool's signal configuration.

20.5.4 Address Space Information

20.5.4.1 `ompd_get_omp_version`

Summary

The tool may call the **`ompd_get_omp_version`** function to obtain the version of the OpenMP API that is associated with an address space.

Format

C

```
ompd_rc_t ompd_get_omp_version(
  ompd_address_space_handle_t *address_space,
  ompd_word_t *omp_version
);
```

C

Semantics

The tool may call the **`ompd_get_omp_version`** function to obtain the version of the OpenMP API that is associated with the address space.

Description of Arguments

The *address_space* argument is an opaque handle that the tool provides to reference the address space of the OpenMP process or device.

Upon return, the *omp_version* argument contains the version of the OpenMP runtime in the **`_OPENMP`** version macro format.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **`ompd_address_space_handle_t`** type, see Section 20.3.8.
- **`ompd_rc_t`** type, see Section 20.3.12.

20.5.4.2 ompd_get_omp_version_string

Summary

The **ompd_get_omp_version_string** function returns a descriptive string for the OpenMP API version that is associated with an address space.

Format

C

```
ompd_rc_t ompd_get_omp_version_string(
  ompd_address_space_handle_t *address_space,
  const char **string
);
```

C

Semantics

After initialization, the tool may call the **ompd_get_omp_version_string** function to obtain the version of the OpenMP API that is associated with an address space.

Description of Arguments

The *address_space* argument is an opaque handle that the tool provides to reference the address space of the OpenMP process or device. A pointer to a descriptive version string is placed into the location to which the *string* output argument points. After returning from the call, the tool owns the string. The OMPD library must use the memory allocation callback that the tool provides to allocate the string storage. The tool is responsible for releasing the memory.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_address_space_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.

20.5.5 Thread Handles

20.5.5.1 ompd_get_thread_in_parallel

Summary

The **ompd_get_thread_in_parallel** function enables a tool to obtain handles for OpenMP threads that are associated with a parallel region.

Format

C

```
ompd_rc_t ompd_get_thread_in_parallel(
  ompd_parallel_handle_t *parallel_handle,
  int thread_num,
  ompd_thread_handle_t **thread_handle
);
```

C

Semantics

A successful invocation of **ompd_get_thread_in_parallel** returns a pointer to a thread handle in the location to which **thread_handle** points. This call yields meaningful results only if all OpenMP threads in the team that is executing the parallel region are stopped.

Description of Arguments

The *parallel_handle* argument is an opaque handle for a parallel region and selects the parallel region on which to operate. The *thread_num* argument selects the thread, the handle of which is to be returned. On return, the *thread_handle* argument is an opaque handle for the selected thread.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_bad_input** if the *thread_num* argument is greater than or equal to the *team-size-var* ICV or negative.

Restrictions

Restrictions on the **ompd_get_thread_in_parallel** function are as follows:

- The value of *thread_num* must be a non-negative integer smaller than the team size that was provided as the *team-size-var* ICV from **ompd_get_icv_from_scope**.

Cross References

- **ompd_parallel_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_thread_handle_t** type, see Section 20.3.8.
- **ompd_get_icv_from_scope** routine, see Section 20.5.10.2.

20.5.5.2 ompd_get_thread_handle

Summary

The **ompd_get_thread_handle** function maps a native thread to an OMPD thread handle.

Format

C

```c
ompd_rc_t ompd_get_thread_handle(
  ompd_address_space_handle_t *handle,
  ompd_thread_id_t kind,
  ompd_size_t sizeof_thread_id,
  const void *thread_id,
  ompd_thread_handle_t **thread_handle
);
```

C

Semantics

The **ompd_get_thread_handle** function determines if the native thread identifier to which *thread_id* points represents an OpenMP thread. If so, the function returns **ompd_rc_ok** and the location to which *thread_handle* points is set to the thread handle for the OpenMP thread.

Description of Arguments

The *handle* argument is an opaque handle that the tool provides to reference an address space. The *kind*, *sizeof_thread_id*, and *thread_id* arguments represent a native thread identifier. On return, the *thread_handle* argument provides an opaque handle to the thread within the provided address space.

The native thread identifier to which *thread_id* points is guaranteed to be valid for the duration of the call. If the OMPD library must retain the native thread identifier, it must copy it.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or any of the following return codes:

- **ompd_rc_bad_input** if a different value in *sizeof_thread_id* is expected for a thread kind of *kind*.
- **ompd_rc_unsupported** if the *kind* of thread is not supported.
- **ompd_rc_unavailable** if the thread is not an OpenMP thread.

Cross References

- **ompd_thread_id_t** type, see Section 20.3.7.
- **ompd_address_space_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_thread_handle_t** type, see Section 20.3.8.
- **ompd_size_t** type, see Section 20.3.1.

20.5.5.3 ompd_rel_thread_handle

Summary

The **ompd_rel_thread_handle** function releases a thread handle.

Format

C

```c
ompd_rc_t ompd_rel_thread_handle(
  ompd_thread_handle_t *thread_handle
);
```

C

Semantics

Thread handles are opaque to tools, which therefore cannot release them directly. Instead, when the tool is finished with a thread handle it must pass it to **ompd_rel_thread_handle** for disposal.

Description of Arguments

The *thread_handle* argument is an opaque handle for a thread to be released.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_thread_handle_t** type, see Section 20.3.8.

20.5.5.4 ompd_thread_handle_compare

Summary

The **ompd_thread_handle_compare** function allows tools to compare two thread handles.

Format

C

```c
ompd_rc_t ompd_thread_handle_compare(
  ompd_thread_handle_t *thread_handle_1,
  ompd_thread_handle_t *thread_handle_2,
  int *cmp_value
);
```

C

Semantics

The internal structure of thread handles is opaque to a tool. While the tool can easily compare pointers to thread handles, it cannot determine whether handles of two different addresses refer to the same underlying thread. The **ompd_thread_handle_compare** function compares thread handles.

On success, **ompd_thread_handle_compare** returns in the location to which *cmp_value* points a signed integer value that indicates how the underlying threads compare: a value less than, equal to, or greater than 0 indicates that the thread corresponding to *thread_handle_1* is, respectively, less than, equal to, or greater than that corresponding to *thread_handle_2*.

Description of Arguments

The *thread_handle_1* and *thread_handle_2* arguments are opaque handles for threads. On return the *cmp_value* argument is set to a signed integer value.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_thread_handle_t** type, see Section 20.3.8.

20.5.5.5 ompd_get_thread_id

Summary

The **ompd_get_thread_id** function maps an OMPD thread handle to a native thread.

Format

C

```
ompd_rc_t ompd_get_thread_id(
  ompd_thread_handle_t *thread_handle,
  ompd_thread_id_t kind,
  ompd_size_t sizeof_thread_id,
  void *thread_id
);
```

C

Semantics

The **ompd_get_thread_id** function maps an OMPD thread handle to a native thread identifier. This call yields meaningful results only if the referenced OpenMP thread is stopped.

Description of Arguments

The *thread_handle* argument is an opaque thread handle. The *kind* argument represents the native thread identifier. The *sizeof_thread_id* argument represents the size of the native thread identifier. On return, the *thread_id* argument is a buffer that represents a native thread identifier.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or any of the following return codes:

- **ompd_rc_bad_input** if a different value in *sizeof_thread_id* is expected for a thread kind of *kind*; or
- **ompd_rc_unsupported** if the *kind* of thread is not supported.

Cross References

- **ompd_thread_id_t** type, see Section 20.3.7.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_thread_handle_t** type, see Section 20.3.8.
- **ompd_size_t** type, see Section 20.3.1.

20.5.5.6 ompd_get_device_from_thread

Summary

The **ompd_get_device_from_thread** function obtains a pointer to the address space handle for a device on which an OpenMP thread is executing.

Format

C

```c
ompd_rc_t ompd_get_device_from_thread(
  ompd_thread_handle_t *thread_handle,
  ompd_address_space_handle_t **device
);
```

C

Semantics

The **ompd_get_device_from_thread** function obtains a pointer to the address space handle for a device on which an OpenMP thread is executing. The returned pointer will be the same as the address space handle pointer that was previously returned by a call to **ompd_process_initialize** (for a host device) or a call to **ompd_device_initialize** (for a non-host device).

This call yields meaningful results only if the referenced OpenMP thread is stopped.

Description of Arguments

The *thread_handle* argument is a pointer to an opaque thread handle that represents an OpenMP thread. On return, the *device* argument is the address of a pointer to an OMPD address space handle.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_address_space_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_thread_handle_t** type, see Section 20.3.8.

20.5.6 Parallel Region Handles

20.5.6.1 ompd_get_curr_parallel_handle

Summary

The **ompd_get_curr_parallel_handle** function obtains a pointer to the parallel handle for an OpenMP thread's current parallel region.

Format

C

```c
ompd_rc_t ompd_get_curr_parallel_handle(
  ompd_thread_handle_t *thread_handle,
  ompd_parallel_handle_t **parallel_handle
);
```

C

Semantics

The **ompd_get_curr_parallel_handle** function enables the tool to obtain a pointer to the parallel handle for the current parallel region that is associated with an OpenMP thread. This call yields meaningful results only if the referenced OpenMP thread is stopped. The parallel handle is owned by the tool and it must be released by calling **ompd_rel_parallel_handle**.

Description of Arguments

The *thread_handle* argument is an opaque handle for a thread and selects the thread on which to operate. On return, the *parallel_handle* argument is set to a handle for the parallel region that the associated thread is currently executing, if any.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_unavailable** if the thread is not currently part of a team.

Cross References

- **ompd_parallel_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_thread_handle_t** type, see Section 20.3.8.
- **ompd_rel_parallel_handle** routine, see Section 20.5.6.4.

20.5.6.2 ompd_get_enclosing_parallel_handle

Summary

The **ompd_get_enclosing_parallel_handle** function obtains a pointer to the parallel handle for an enclosing parallel region.

Format

C

```
ompd_rc_t ompd_get_enclosing_parallel_handle(
  ompd_parallel_handle_t *parallel_handle,
  ompd_parallel_handle_t **enclosing_parallel_handle
);
```

C

Semantics

The **ompd_get_enclosing_parallel_handle** function enables a tool to obtain a pointer to the parallel handle for the parallel region that encloses the parallel region that **parallel_handle** specifies. This call is meaningful only if at least one thread in the team that is executing the parallel region is stopped. A pointer to the parallel handle for the enclosing region is returned in the location to which *enclosing_parallel_handle* points. After the call, the tool owns the handle; the tool must release the handle with **ompd_rel_parallel_handle** when it is no longer required.

Description of Arguments

The *parallel_handle* argument is an opaque handle for a parallel region that selects the parallel region on which to operate. On return, the *enclosing_parallel_handle* argument is set to a handle for the parallel region that encloses the selected parallel region.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_unavailable** if no enclosing parallel region exists.

Cross References

- **ompd_parallel_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_rel_parallel_handle** routine, see Section 20.5.6.4.

20.5.6.3 ompd_get_task_parallel_handle

Summary

The **ompd_get_task_parallel_handle** function obtains a pointer to the parallel handle for the parallel region that encloses a task region.

Format

C

```c
ompd_rc_t ompd_get_task_parallel_handle(
  ompd_task_handle_t *task_handle,
  ompd_parallel_handle_t **task_parallel_handle
);
```

C

Semantics

The **ompd_get_task_parallel_handle** function enables a tool to obtain a pointer to the parallel handle for the parallel region that encloses the task region that *task_handle* specifies. This call yields meaningful results only if at least one thread in the team that is executing the parallel region is stopped. A pointer to the parallel regions handle is returned in the location to which *task_parallel_handle* points. The tool owns that parallel handle, which it must release with **ompd_rel_parallel_handle**.

Description of Arguments

The *task_handle* argument is an opaque handle that selects the task on which to operate. On return, the *parallel_handle* argument is set to a handle for the parallel region that encloses the selected task.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_parallel_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_task_handle_t** type, see Section 20.3.8.
- **ompd_rel_parallel_handle** routine, see Section 20.5.6.4.

20.5.6.4 ompd_rel_parallel_handle

Summary

The **ompd_rel_parallel_handle** function releases a parallel region handle.

Format

C

```
ompd_rc_t ompd_rel_parallel_handle(
  ompd_parallel_handle_t *parallel_handle
);
```

C

Semantics

Parallel region handles are opaque so tools cannot release them directly. Instead, a tool must pass a parallel region handle to the **ompd_rel_parallel_handle** function for disposal when finished with it.

Description of Arguments

The *parallel_handle* argument is an opaque handle to be released.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_parallel_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.

20.5.6.5 ompd_parallel_handle_compare

Summary

The **ompd_parallel_handle_compare** function compares two parallel region handles.

Format

C

```c
ompd_rc_t ompd_parallel_handle_compare(
  ompd_parallel_handle_t *parallel_handle_1,
  ompd_parallel_handle_t *parallel_handle_2,
  int *cmp_value
);
```

C

Semantics

The internal structure of parallel region handles is opaque to tools. While tools can easily compare pointers to parallel region handles, they cannot determine whether handles at two different addresses refer to the same underlying parallel region and, instead must use the **ompd_parallel_handle_compare** function.

On success, **ompd_parallel_handle_compare** returns a signed integer value in the location to which *cmp_value* points that indicates how the underlying parallel regions compare. A value less than, equal to, or greater than 0 indicates that the region corresponding to *parallel_handle_1* is, respectively, less than, equal to, or greater than that corresponding to *parallel_handle_2*. This function is provided since the means by which parallel region handles are ordered is implementation defined.

Description of Arguments

The *parallel_handle_1* and *parallel_handle_2* arguments are opaque handles that correspond to parallel regions. On return the *cmp_value* argument points to a signed integer value that indicates how the underlying parallel regions compare.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_parallel_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.

20.5.7 Task Handles

20.5.7.1 ompd_get_curr_task_handle

Summary

The **ompd_get_curr_task_handle** function obtains a pointer to the task handle for the current task region that is associated with an OpenMP thread.

Format

C

```
ompd_rc_t ompd_get_curr_task_handle(
  ompd_thread_handle_t *thread_handle,
  ompd_task_handle_t **task_handle
);
```

C

Semantics

The **ompd_get_curr_task_handle** function obtains a pointer to the task handle for the current task region that is associated with an OpenMP thread. This call yields meaningful results only if the thread for which the handle is provided is stopped. The task handle must be released with **ompd_rel_task_handle**.

Description of Arguments

The *thread_handle* argument is an opaque handle that selects the thread on which to operate. On return, the *task_handle* argument points to a location that points to a handle for the task that the thread is currently executing.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_unavailable** if the thread is currently not executing a task.

Cross References

- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_task_handle_t** type, see Section 20.3.8.
- **ompd_thread_handle_t** type, see Section 20.3.8.
- **ompd_rel_task_handle** routine, see Section 20.5.7.5.

20.5.7.2 ompd_get_generating_task_handle

Summary

The **ompd_get_generating_task_handle** function obtains a pointer to the task handle of the generating task region.

Format

C

```
ompd_rc_t ompd_get_generating_task_handle(
  ompd_task_handle_t *task_handle,
  ompd_task_handle_t **generating_task_handle
);
```

C

Semantics

The **ompd_get_generating_task_handle** function obtains a pointer to the task handle for the task that encountered the OpenMP task construct that generated the task represented by *task_handle*. The generating task is the OpenMP task that was active when the task specified by *task_handle* was created. This call yields meaningful results only if the thread that is executing the task that *task_handle* specifies is stopped while executing the task. The generating task handle must be released with **ompd_rel_task_handle**.

Description of Arguments

The *task_handle* argument is an opaque handle that selects the task on which to operate. On return, the *generating_task_handle* argument points to a location that points to a handle for the generating task.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_unavailable** if no generating task region exists.

Cross References

- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_task_handle_t** type, see Section 20.3.8.
- **ompd_rel_task_handle** routine, see Section 20.5.7.5.

20.5.7.3 ompd_get_scheduling_task_handle

Summary

The **ompd_get_scheduling_task_handle** function obtains a task handle for the task that was active at a task scheduling point.

Format

C

```
ompd_rc_t ompd_get_scheduling_task_handle(
  ompd_task_handle_t *task_handle,
  ompd_task_handle_t **scheduling_task_handle
);
```

C

Semantics

The **ompd_get_scheduling_task_handle** function obtains a task handle for the task that was active when the task that *task_handle* represents was scheduled. An implicit task does not have a scheduling task. This call yields meaningful results only if the thread that is executing the task that *task_handle* specifies is stopped while executing the task. The scheduling task handle must be released with **ompd_rel_task_handle**.

Description of Arguments

The *task_handle* argument is an opaque handle for a task and selects the task on which to operate. On return, the *scheduling_task_handle* argument points to a location that points to a handle for the task that is still on the stack of execution on the same thread and was deferred in favor of executing the selected task.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_unavailable** if no scheduling task exists.

Cross References

- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_task_handle_t** type, see Section 20.3.8.
- **ompd_rel_task_handle** routine, see Section 20.5.7.5.

20.5.7.4 ompd_get_task_in_parallel

Summary

The **ompd_get_task_in_parallel** function obtains handles for the implicit tasks that are associated with a parallel region.

Format

C

```c
ompd_rc_t ompd_get_task_in_parallel(
  ompd_parallel_handle_t *parallel_handle,
  int thread_num,
  ompd_task_handle_t **task_handle
);
```

C

Semantics

The **ompd_get_task_in_parallel** function obtains handles for the implicit tasks that are associated with a parallel region. A successful invocation of **ompd_get_task_in_parallel** returns a pointer to a task handle in the location to which *task_handle* points. This call yields meaningful results only if all OpenMP threads in the parallel region are stopped.

Description of Arguments

The *parallel_handle* argument is an opaque handle that selects the parallel region on which to operate. The *thread_num* argument selects the implicit task of the team to be returned. The *thread_num* argument is equal to the *thread-num-var* ICV value of the selected implicit task. On return, the *task_handle* argument points to a location that points to an opaque handle for the selected implicit task.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_bad_input** if the *thread_num* argument is greater than or equal to the *team-size-var* ICV or negative.

Restrictions

Restrictions on the **ompd_get_task_in_parallel** function are as follows:

- The value of *thread_num* must be a non-negative integer that is smaller than the size of the team size that is the value of the *team-size-var* ICV that **ompd_get_icv_from_scope** returns.

Cross References

- **ompd_parallel_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_task_handle_t** type, see Section 20.3.8.
- **ompd_get_icv_from_scope** routine, see Section 20.5.10.2.

20.5.7.5 `ompd_rel_task_handle`

Summary

This **`ompd_rel_task_handle`** function releases a task handle.

Format

C

```c
ompd_rc_t ompd_rel_task_handle(
  ompd_task_handle_t *task_handle
);
```

C

Semantics

Task handles are opaque to tools; thus tools cannot release them directly. Instead, when a tool is finished with a task handle it must use the **`ompd_rel_task_handle`** function to release it.

Description of Arguments

The *task_handle* argument is an opaque task handle to be released.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **`ompd_rc_t`** type, see Section 20.3.12.
- **`ompd_task_handle_t`** type, see Section 20.3.8.

20.5.7.6 `ompd_task_handle_compare`

Summary

The **`ompd_task_handle_compare`** function compares task handles.

Format

C

```c
ompd_rc_t ompd_task_handle_compare(
  ompd_task_handle_t *task_handle_1,
  ompd_task_handle_t *task_handle_2,
  int *cmp_value
);
```

C

Semantics

The internal structure of task handles is opaque; so tools cannot directly determine if handles at two different addresses refer to the same underlying task. The **ompd_task_handle_compare** function compares task handles. After a successful call to **ompd_task_handle_compare**, the value of the location to which *cmp_value* points is a signed integer that indicates how the underlying tasks compare: a value less than, equal to, or greater than 0 indicates that the task that corresponds to *task_handle_1* is, respectively, less than, equal to, or greater than the task that corresponds to *task_handle_2*. The means by which task handles are ordered is implementation defined.

Description of Arguments

The *task_handle_1* and *task_handle_2* arguments are opaque handles that correspond to tasks. On return, the *cmp_value* argument points to a location in which a signed integer value indicates how the underlying tasks compare.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_task_handle_t** type, see Section 20.3.8.

20.5.7.7 ompd_get_task_function

Summary

This **ompd_get_task_function** function returns the entry point of the code that corresponds to the body of a task.

Format

C

```c
ompd_rc_t ompd_get_task_function (
  ompd_task_handle_t *task_handle,
  ompd_address_t *entry_point
);
```

C

Semantics

The **ompd_get_task_function** function returns the entry point of the code that corresponds to the body of code that the task executes. This call is meaningful only if the thread that is executing the task that *task_handle* specifies is stopped whle executing the task.

Description of Arguments

The *task_handle* argument is an opaque handle that selects the task on which to operate. On return, the *entry_point* argument is set to an address that describes the beginning of application code that executes the task region.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_task_handle_t** type, see Section 20.3.8.
- **ompd_address_t** type, see Section 20.3.4.

20.5.7.8 ompd_get_task_frame

Summary

The **ompd_get_task_frame** function extracts the frame pointers of a task.

Format

C

```
ompd_rc_t ompd_get_task_frame (
  ompd_task_handle_t *task_handle,
  ompd_frame_info_t *exit_frame,
  ompd_frame_info_t *enter_frame
);
```

C

Semantics

An OpenMP implementation maintains an **ompt_frame_t** object for every implicit or explicit task. The **ompd_get_task_frame** function extracts the *enter_frame* and *exit_frame* fields of the **ompt_frame_t** object of the task that *task_handle* identifies. This call yields meaningful results only if the thread that is executing the task that *task_handle* specifies is stopped while executing the task.

Description of Arguments

The *task_handle* argument specifies an OpenMP task. On return, the *exit_frame* argument points to an **ompd_frame_info_t** object that has the frame information with the same semantics as the *exit_frame* field in the **ompt_frame_t** object that is associated with the specified task. On return, the *enter_frame* argument points to an **ompd_frame_info_t** object that has the frame information with the same semantics as the *enter_frame* field in the **ompt_frame_t** object that is associated with the specified task.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompt_frame_t** type, see Section 19.4.4.29.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_task_handle_t** type, see Section 20.3.8.
- **ompd_address_t** type, see Section 20.3.4.
- **ompd_frame_info_t** type, see Section 20.3.5.

20.5.8 Querying Thread States

20.5.8.1 ompd_enumerate_states

Summary

The **ompd_enumerate_states** function enumerates thread states that an OpenMP implementation supports.

Format

C

```c
ompd_rc_t ompd_enumerate_states (
  ompd_address_space_handle_t *address_space_handle,
  ompd_word_t current_state,
  ompd_word_t *next_state,
  const char **next_state_name,
  ompd_word_t *more_enums
);
```

C

Semantics

An OpenMP implementation may support only a subset of the states that the **ompt_state_t** enumeration type defines. In addition, an OpenMP implementation may support implementation-specific states. The **ompd_enumerate_states** call enables a tool to enumerate the thread states that an OpenMP implementation supports.

When the *current_state* argument is a thread state that an OpenMP implementation supports, the call assigns the value and string name of the next thread state in the enumeration to the locations to which the *next_state* and *next_state_name* arguments point.

On return, the third-party tool owns the *next_state_name* string. The OMPD library allocates storage for the string with the memory allocation callback that the tool provides. The tool is responsible for releasing the memory.

On return, the location to which the *more_enums* argument points has the value 1 whenever one or more states are left in the enumeration. On return, the location to which the *more_enums* argument points has the value 0 when *current_state* is the last state in the enumeration.

Description of Arguments

The *address_space_handle* argument identifies the address space. The *current_state* argument must be a thread state that the OpenMP implementation supports. To begin enumerating the supported states, a tool should pass **ompt_state_undefined** as the value of *current_state*. Subsequent calls to **ompd_enumerate_states** by the tool should pass the value that the call returned in the *next_state* argument. On return, the *next_state* argument points to an integer with the value of the next state in the enumeration. On return, the *next_state_name* argument points to a character string that describes the next state. On return, the *more_enums* argument points to an integer with a value of 1 when more states are left to enumerate and a value of 0 when no more states are left.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_bad_input** if an unknown value is provided in *current_state*.

Cross References

- **ompt_state_t** type, see Section 19.4.4.28.
- **ompd_address_space_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.

20.5.8.2 ompd_get_state

Summary

The **ompd_get_state** function obtains the state of a thread.

Format

C

```
ompd_rc_t ompd_get_state (
  ompd_thread_handle_t *thread_handle,
  ompd_word_t *state,
  ompd_wait_id_t *wait_id
);
```

C

Semantics

The **ompd_get_state** function returns the state of an OpenMP thread. This call yields meaningful results only if the referenced OpenMP thread is stopped.

Description of Arguments

The *thread_handle* argument identifies the thread. The *state* argument represents the state of that thread as represented by a value that **ompd_enumerate_states** returns. On return, if the *wait_id* argument is non-null then it points to a handle that corresponds to the *wait_id* wait identifier of the thread. If the thread state is not one of the specified wait states, the value to which *wait_id* points is undefined.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_thread_handle_t** type, see Section 20.3.8.
- **ompd_enumerate_states** routine, see Section 20.5.8.1.
- **ompd_wait_id_t** type, see Section 20.3.2.

20.5.9 Display Control Variables

20.5.9.1 ompd_get_display_control_vars

Summary

The **ompd_get_display_control_vars** function returns a list of name/value pairs for OpenMP control variables.

Format

C

```
ompd_rc_t ompd_get_display_control_vars (
  ompd_address_space_handle_t *address_space_handle,
  const char * const **control_vars
);
```

C

Semantics

The **ompd_get_display_control_vars** function returns a *NULL*-terminated vector of null-terminated strings of name/value pairs of control variables that have user controllable settings and are important to the operation or performance of an OpenMP runtime system. The control variables that this interface exposes include all OpenMP environment variables, settings that may come from vendor or platform-specific environment variables, and other settings that affect the operation or functioning of an OpenMP runtime.

The format of the strings is **"icv-name=icv-value"**.

On return, the third-party tool owns the vector and the strings. The OMPD library must satisfy the termination constraints; it may use static or dynamic memory for the vector and/or the strings and is unconstrained in how it arranges them in memory. If it uses dynamic memory then the OMPD library must use the allocate callback that the tool provides to **ompd_initialize**. The tool must use the **ompd_rel_display_control_vars** function to release the vector and the strings.

Description of Arguments

The *address_space_handle* argument identifies the address space. On return, the *control_vars* argument points to the vector of display control variables.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_address_space_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_initialize** routine, see Section 20.5.1.1.
- **ompd_rel_display_control_vars** routine, see Section 20.5.9.2.

20.5.9.2 ompd_rel_display_control_vars

Summary

The **ompd_rel_display_control_vars** releases a list of name/value pairs of OpenMP control variables previously acquired with **ompd_get_display_control_vars**.

Format

C

```
ompd_rc_t ompd_rel_display_control_vars (
  const char * const **control_vars
);
```

C

Semantics

The third-party tool owns the vector and strings that **ompd_get_display_control_vars** returns. The tool must call **ompd_rel_display_control_vars** to release the vector and the strings.

Description of Arguments

The *control_vars* argument is the vector of display control variables to be released.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5.

Cross References

- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_get_display_control_vars** routine, see Section 20.5.9.1.

20.5.10 Accessing Scope-Specific Information

20.5.10.1 ompd_enumerate_icvs

Summary

The **ompd_enumerate_icvs** function enumerates ICVs.

Format

C

```c
ompd_rc_t ompd_enumerate_icvs (
  ompd_address_space_handle_t *handle,
  ompd_icv_id_t current,
  ompd_icv_id_t *next_id,
  const char **next_icv_name,
  ompd_scope_t *next_scope,
  int *more
);
```

C

Semantics

An OpenMP implementation must support all ICVs listed in Section 2.1. An OpenMP implementation may support additional implementation-specific variables. An implementation may store ICVs in a different scope than Table 2.3 indicates. The **ompd_enumerate_icvs** function enables a tool to enumerate the ICVs that an OpenMP implementation supports and their related scopes. The ICVs *num-procs-var*, *thread-num-var*, *final-task-var*, *implicit-task-var* and *team-size-var* must also be available with an *ompd-* prefix.

When the *current* argument is set to the identifier of a supported ICV, **ompd_enumerate_icvs** assigns the value, string name, and scope of the next ICV in the enumeration to the locations to which the *next_id*, *next_icv_name*, and *next_scope* arguments point. On return, the third-party tool owns the *next_icv_name* string. The OMPD library uses the memory allocation callback that the tool provides to allocate the string storage; the tool is responsible for releasing the memory.

On return, the location to which the *more* argument points has the value of 1 whenever one or more ICV are left in the enumeration. On return, that location has the value 0 when *current* is the last ICV in the enumeration.

Description of Arguments

The *address_space_handle* argument identifies the address space. The *current* argument must be an ICV that the OpenMP implementation supports. To begin enumerating the ICVs, a tool should pass **ompd_icv_undefined** as the value of *current*. Subsequent calls to **ompd_enumerate_icvs** should pass the value returned by the call in the *next_id* output argument. On return, the *next_id* argument points to an integer with the value of the ID of the next ICV in the enumeration. On return, the *next_icv_name* argument points to a character string with the name of the next ICV. On return, the *next_scope* argument points to the scope enum value of the scope of the next ICV. On return, the *more_enums* argument points to an integer with the value of 1 when more ICVs are left to enumerate and the value of 0 when no more ICVs are left.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_bad_input** if an unknown value is provided in *current*.

Cross References

- **ompd_address_space_handle_t** type, see Section 20.3.8.
- **ompd_icv_id_t** type, see Section 20.3.10.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_scope_t** type, see Section 20.3.9.

20.5.10.2 `ompd_get_icv_from_scope`

Summary

The **`ompd_get_icv_from_scope`** function returns the value of an ICV.

Format

C

```
ompd_rc_t ompd_get_icv_from_scope (
  void *handle,
  ompd_scope_t scope,
  ompd_icv_id_t icv_id,
  ompd_word_t *icv_value
);
```

C

Semantics

The **`ompd_get_icv_from_scope`** function provides access to the ICVs that **`ompd_enumerate_icvs`** identifies.

Description of Arguments

The *handle* argument provides an OpenMP scope handle. The *scope* argument specifies the kind of scope provided in *handle*. The *icv_id* argument specifies the ID of the requested ICV. On return, the *icv_value* argument points to a location with the value of the requested ICV.

Constraints on Arguments

The provided *handle* must match the *scope* as defined in Section 20.3.10.

The provided *scope* must match the scope for *icv_id* as requested by **`ompd_enumerate_icvs`**.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or any of the following return codes:

- **`ompd_rc_incompatible`** if the ICV cannot be represented as an integer;
- **`ompd_rc_incomplete`** if only the first item of the ICV is returned in the integer (e.g., if *nthreads-var* is a list); or
- **`ompd_rc_bad_input`** if an unknown value is provided in *icv_id*.

Cross References

- **ompd_address_space_handle_t** type, see Section 20.3.8.
- **ompd_icv_id_t** type, see Section 20.3.10.
- **ompd_parallel_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_scope_t** type, see Section 20.3.9.
- **ompd_task_handle_t** type, see Section 20.3.8.
- **ompd_thread_handle_t** type, see Section 20.3.8.
- **ompd_enumerate_icvs** routine, see Section 20.5.10.1.

20.5.10.3 ompd_get_icv_string_from_scope

Summary

The **ompd_get_icv_string_from_scope** function returns the value of an ICV.

Format

C

```
ompd_rc_t ompd_get_icv_string_from_scope (
  void *handle,
  ompd_scope_t scope,
  ompd_icv_id_t icv_id,
  const char **icv_string
);
```

C

Semantics

The **ompd_get_icv_string_from_scope** function provides access to the ICVs that **ompd_enumerate_icvs** identifies.

Description of Arguments

The *handle* argument provides an OpenMP scope handle. The *scope* argument specifies the kind of scope provided in *handle*. The *icv_id* argument specifies the ID of the requested ICV. On return, the *icv_string* argument points to a string representation of the requested ICV.

On return, the third-party tool owns the *icv_string* string. The OMPD library allocates the string storage with the memory allocation callback that the tool provides. The tool is responsible for releasing the memory.

Constraints on Arguments

The provided *handle* must match the *scope* as defined in Section 20.3.10.

The provided *scope* must match the scope for *icv_id* as requested by **ompd_enumerate_icvs**.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_bad_input** if an unknown value is provided in *icv_id*.

Cross References

- **ompd_address_space_handle_t** type, see Section 20.3.8.
- **ompd_icv_id_t** type, see Section 20.3.10.
- **ompd_parallel_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_scope_t** type, see Section 20.3.9.
- **ompd_task_handle_t** type, see Section 20.3.8.
- **ompd_thread_handle_t** type, see Section 20.3.8.
- **ompd_enumerate_icvs** routine, see Section 20.5.10.1.

20.5.10.4 ompd_get_tool_data

Summary

The **ompd_get_tool_data** function provides access to the OMPT data variable stored for each OpenMP scope.

Format

C

```
ompd_rc_t ompd_get_tool_data(
  void* handle,
  ompd_scope_t scope,
  ompd_word_t *value,
  ompd_address_t *ptr
);
```

C

Semantics

The **ompd_get_tool_data** function provides access to the OMPT tool data stored for each scope. If the runtime library does not support OMPT then the function returns **ompd_rc_unsupported**.

Description of Arguments

The *handle* argument provides an OpenMP scope handle. The *scope* argument specifies the kind of scope provided in *handle*. On return, the *value* argument points to the *value* field of the **ompt_data_t** union stored for the selected scope. On return, the *ptr* argument points to the *ptr* field of the **ompt_data_t** union stored for the selected scope.

Description of Return Codes

This routine must return any of the general return codes listed at the beginning of Section 20.5 or the following return code:

- **ompd_rc_unsupported** if the runtime library does not support OMPT.

Cross References

- **ompt_data_t** type, see Section 19.4.4.4.
- **ompd_address_space_handle_t** type, see Section 20.3.8.
- **ompd_parallel_handle_t** type, see Section 20.3.8.
- **ompd_rc_t** type, see Section 20.3.12.
- **ompd_scope_t** type, see Section 20.3.9.
- **ompd_task_handle_t** type, see Section 20.3.8.
- **ompd_thread_handle_t** type, see Section 20.3.8.

20.6 Runtime Entry Points for OMPD

The OpenMP implementation must define several entry point symbols through which execution must pass when particular events occur *and* data collection for OMPD is enabled. A tool can enable notification of an event by setting a breakpoint at the address of the entry point symbol.

Entry point symbols have external **C** linkage and do not require demangling or other transformations to look up their names to obtain the address in the OpenMP program. While each entry point symbol conceptually has a function type signature, it may not be a function. It may be a labeled location

20.6.1 Beginning Parallel Regions

Summary

Before starting the execution of an OpenMP parallel region, the implementation executes **ompd_bp_parallel_begin**.

Format

C

```c
void ompd_bp_parallel_begin(void);
```

C

Semantics

The OpenMP implementation must execute **ompd_bp_parallel_begin** at every *parallel-begin* event. At the point that the implementation reaches **ompd_bp_parallel_begin**, the binding for **ompd_get_curr_parallel_handle** is the parallel region that is beginning and the binding for **ompd_get_curr_task_handle** is the task that encountered the **parallel** construct.

Cross References

- **parallel** construct, see Section 10.1.
- **ompd_get_curr_parallel_handle** routine, see Section 20.5.6.1.
- **ompd_get_curr_task_handle** routine, see Section 20.5.7.1.

20.6.2 Ending Parallel Regions

Summary

After finishing the execution of an OpenMP parallel region, the implementation executes **ompd_bp_parallel_end**.

Format

C

```c
void ompd_bp_parallel_end(void);
```

C

Semantics

The OpenMP implementation must execute **ompd_bp_parallel_end** at every *parallel-end* event. At the point that the implementation reaches **ompd_bp_parallel_end**, the binding for **ompd_get_curr_parallel_handle** is the **parallel** region that is ending and the binding for **ompd_get_curr_task_handle** is the task that encountered the **parallel** construct. After execution of **ompd_bp_parallel_end**, any *parallel_handle* that was acquired for the **parallel** region is invalid and should be released.

Cross References

- **parallel** construct, see Section 10.1.
- **ompd_get_curr_parallel_handle** routine, see Section 20.5.6.1.
- **ompd_get_curr_task_handle** routine, see Section 20.5.7.1.
- **ompd_rel_parallel_handle** routine, see Section 20.5.6.4.

20.6.3 Beginning Task Regions

Summary

Before starting the execution of an OpenMP task region, the implementation executes **ompd_bp_task_begin**.

Format

C

```
void ompd_bp_task_begin(void);
```

C

Semantics

The OpenMP implementation must execute **ompd_bp_task_begin** immediately before starting execution of a *structured-block* that is associated with a non-merged task. At the point that the implementation reaches **ompd_bp_task_begin**, the binding for **ompd_get_curr_task_handle** is the task that is scheduled to execute.

Cross References

- **ompd_get_curr_task_handle** routine, see Section 20.5.7.1.

20.6.4 Ending Task Regions

Summary

After finishing the execution of an OpenMP task region, the implementation executes **ompd_bp_task_end**.

Format

C

```
void ompd_bp_task_end(void);
```

C

Semantics

The OpenMP implementation must execute **ompd_bp_task_end** immediately after completion of a *structured-block* that is associated with a non-merged task. At the point that the implementation reaches **ompd_bp_task_end**, the binding for **ompd_get_curr_task_handle** is the task that finished execution. After execution of **ompd_bp_task_end**, any *task_handle* that was acquired for the task region is invalid and should be released.

Cross References

- **ompd_get_curr_task_handle** routine, see Section 20.5.7.1.
- **ompd_rel_task_handle** routine, see Section 20.5.7.5.

20.6.5 Beginning OpenMP Threads

Summary

When starting an OpenMP thread, the implementation executes **ompd_bp_thread_begin**.

Format

C

```
void ompd_bp_thread_begin(void);
```

C

Semantics

The OpenMP implementation must execute **ompd_bp_thread_begin** at every *native-thread-begin* and *initial-thread-begin* event. This execution occurs before the thread starts the execution of any OpenMP region.

Cross References

- **parallel** construct, see Section 10.1.
- Initial task, see Section 12.8.

20.6.6 Ending OpenMP Threads

Summary

When terminating an OpenMP thread, the implementation executes **ompd_bp_thread_end**.

Format

C

```
void ompd_bp_thread_end(void);
```

C

Semantics

The OpenMP implementation must execute **ompd_bp_thread_end** at every *native-thread-end* and *initial-thread-end* event. This execution occurs after the thread completes the execution of all OpenMP regions. After executing **ompd_bp_thread_end**, any *thread_handle* that was acquired for this thread is invalid and should be released.

Cross References

- **parallel** construct, see Section 10.1.
- Initial task, see Section 12.8.
- **ompd_rel_thread_handle** routine, see Section 20.5.5.3.

20.6.7 Initializing OpenMP Devices

Summary

The OpenMP implementation must execute **ompd_bp_device_begin** at every *device-initialize* event.

Format

C

```
void ompd_bp_device_begin(void);
```

C

Semantics

When initializing a device for execution of a **target** region, the implementation must execute **ompd_bp_device_begin**. This execution occurs before the work associated with any OpenMP region executes on the device.

Cross References

- Device Initialization, see Section 13.4.

20.6.8 Finalizing OpenMP Devices

Summary

When terminating an OpenMP thread, the implementation executes **ompd_bp_device_end**.

Format

C

```
void ompd_bp_device_end(void);
```

C

Semantics

The OpenMP implementation must execute **ompd_bp_device_end** at every *device-finalize* event. This execution occurs after the thread executes all OpenMP regions. After execution of **ompd_bp_device_end**, any *address_space_handle* that was acquired for this device is invalid and should be released.

Cross References

- Device Initialization, see Section 13.4.
- **ompd_rel_address_space_handle** routine, see Section 20.5.2.3.

21 Environment Variables

This chapter describes the OpenMP environment variables that specify the settings of the ICVs that affect the execution of OpenMP programs (see Section 2). The names of the environment variables must be upper case. Unless otherwise specified, the values assigned to the environment variables are case insensitive and may have leading and trailing white space. Modifications to the environment variables after the program has started, even if modified by the program itself, are ignored by the OpenMP implementation. However, the settings of some of the ICVs can be modified during the execution of the OpenMP program by the use of the appropriate directive clauses or OpenMP API routines.

The following examples demonstrate how the OpenMP environment variables can be set in different environments:

- csh-like shells:

```
setenv OMP_SCHEDULE "dynamic"
```

- bash-like shells:

```
export OMP_SCHEDULE="dynamic"
```

- Windows Command Line:

```
set OMP_SCHEDULE=dynamic
```

As defined following Table 2.1 in Section 2.2, device-specific environment variables extend many of the environment variables defined in this chapter. If the corresponding environment variable for a specific device number, including the host device, is set, then the setting for that environment variable is used to set the value of the associated ICV of the device with the corresponding device number. If the corresponding environment variable that includes the **_DEV** suffix but no device number is set, then the setting of that environment variable is used to set the value of the associated ICV of any non-host device for which the device-number-specific corresponding environment variable is not set. In all cases the setting of an environment variable for which a device number is specified takes precedence.

Restrictions

Restrictions to device-specific environment variables are as follows:

- Device-specific environment variables must not correspond to environment variables that initialize ICVs with global scope.

21.1 Parallel Region Environment Variables

This section defines environment variables that affect the operation of **parallel** regions.

21.1.1 OMP_DYNAMIC

The **OMP_DYNAMIC** environment variable controls dynamic adjustment of the number of threads to use for executing **parallel** regions by setting the initial value of the *dyn-var* ICV.

The value of this environment variable must be one of the following:

true | false

If the environment variable is set to **true**, the OpenMP implementation may adjust the number of threads to use for executing **parallel** regions in order to optimize the use of system resources. If the environment variable is set to **false**, the dynamic adjustment of the number of threads is disabled. The behavior of the program is implementation defined if the value of **OMP_DYNAMIC** is neither **true** nor **false**.

Example:

```
setenv OMP_DYNAMIC true
```

Cross References

- *dyn-var* ICV, see Section 2.
- **omp_get_dynamic** routine, see Section 18.2.7.
- **omp_set_dynamic** routine, see Section 18.2.6.

21.1.2 OMP_NUM_THREADS

The **OMP_NUM_THREADS** environment variable sets the number of threads to use for **parallel** regions by setting the initial value of the *nthreads-var* ICV. See Section 2 for a comprehensive set of rules about the interaction between the **OMP_NUM_THREADS** environment variable, the **num_threads** clause, the **omp_set_num_threads** library routine and dynamic adjustment of threads, and Section 10.1.1 for a complete algorithm that describes how the number of threads for a **parallel** region is determined.

The value of this environment variable must be a list of positive integer values. The values of the list set the number of threads to use for **parallel** regions at the corresponding nested levels.

The behavior of the program is implementation defined if any value of the list specified in the **OMP_NUM_THREADS** environment variable leads to a number of threads that is greater than an implementation can support, or if any value is not a positive integer.

The **OMP_NUM_THREADS** environment variable sets the *max-active-levels-var* ICV to the number of active levels of parallelism that the implementation supports if the **OMP_NUM_THREADS** environment variable is set to a comma-separated list of more than one value. The value of the *max-active-level-var* ICV may be overridden by setting **OMP_MAX_ACTIVE_LEVELS** or **OMP_NESTED**. See Section 21.1.4 and Section 21.1.5 for details.

Example:

```
setenv OMP_NUM_THREADS 4,3,2
```

Cross References

- *nthreads-var* ICV, see Section 2.
- **num_threads** clause, see Section 10.1.
- **omp_get_max_threads** routine, see Section 18.2.3.
- **omp_get_num_threads** routine, see Section 18.2.2.
- **omp_get_team_size** routine, see Section 18.2.19.
- **omp_set_num_threads** routine, see Section 18.2.1.

21.1.3 OMP_THREAD_LIMIT

The **OMP_THREAD_LIMIT** environment variable sets the maximum number of OpenMP threads to use in a contention group by setting the *thread-limit-var* ICV.

The value of this environment variable must be a positive integer. The behavior of the program is implementation defined if the requested value of **OMP_THREAD_LIMIT** is greater than the number of threads an implementation can support, or if the value is not a positive integer.

Cross References

- *thread-limit-var* ICV, see Section 2.
- **omp_get_thread_limit** routine, see Section 18.2.13.

21.1.4 OMP_MAX_ACTIVE_LEVELS

The **OMP_MAX_ACTIVE_LEVELS** environment variable controls the maximum number of nested active **parallel** regions by setting the initial value of the *max-active-levels-var* ICV.

The value of this environment variable must be a non-negative integer. The behavior of the program is implementation defined if the requested value of **OMP_MAX_ACTIVE_LEVELS** is greater than the maximum number of nested active parallel levels an implementation can support, or if the value is not a non-negative integer.

Cross References

- *max-active-levels-var* ICV, see Section 2.
- **omp_get_max_active_levels** routine, see Section 18.2.16.
- **omp_set_max_active_levels** routine, see Section 18.2.15.

21.1.5 **OMP_NESTED** (Deprecated)

The **OMP_NESTED** environment variable controls nested parallelism by setting the initial value of the *max-active-levels-var* ICV. If the environment variable is set to **true**, the initial value of *max-active-levels-var* is set to the number of active levels of parallelism supported by the implementation. If the environment variable is set to **false**, the initial value of *max-active-levels-var* is set to 1. The behavior of the program is implementation defined if the value of **OMP_NESTED** is neither **true** nor **false**.

If both the **OMP_NESTED** and **OMP_MAX_ACTIVE_LEVELS** environment variables are set, the value of **OMP_NESTED** is **false**, and the value of **OMP_MAX_ACTIVE_LEVELS** is greater than 1, then the behavior is implementation defined. Otherwise, if both environment variables are set then the **OMP_NESTED** environment variable has no effect.

The **OMP_NESTED** environment variable has been deprecated.

Example:

```
setenv OMP_NESTED false
```

Cross References

- *max-active-levels-var* ICV, see Section 2.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 21.1.4.
- **omp_get_team_size** routine, see Section 18.2.19.
- **omp_set_nested** routine, see Section 18.2.9.

21.1.6 **OMP_PLACES**

The **OMP_PLACES** environment variable sets the initial value of the *place-partition-var* ICV. A list of places can be specified in the **OMP_PLACES** environment variable. The value of **OMP_PLACES** can be one of two types of values: either an abstract name that describes a set of places or an explicit list of places described by non-negative numbers.

The **OMP_PLACES** environment variable can be defined using an explicit ordered list of comma-separated places. A place is defined by an unordered set of comma-separated non-negative numbers enclosed by braces, or a non-negative number. The meaning of the numbers and how the numbering is done are implementation defined. Generally, the numbers represent the smallest unit of execution exposed by the execution environment, typically a hardware thread.

Intervals may also be used to define places. Intervals can be specified using the <*lower-bound*> : <*length*> : <*stride*> notation to represent the following list of numbers: “<*lower-bound*>, <*lower-bound*> + <*stride*>, ..., <*lower-bound*> + (<*length*> - 1)*<*stride*>.” When <*stride*> is omitted, a unit stride is assumed. Intervals can specify numbers within a place as well as sequences of places.

An exclusion operator "**!**" can also be used to exclude the number or place immediately following the operator.

Alternatively, the abstract names listed in Table 21.1 should be understood by the execution and runtime environment. The precise definitions of the abstract names are implementation defined. An implementation may also add abstract names as appropriate for the target platform.

The abstract name may be appended by a positive number in parentheses to denote the length of the place list to be created, that is *abstract_name(num-places)*. When requesting fewer places than available on the system, the determination of which resources of type *abstract_name* are to be included in the place list is implementation defined. When requesting more resources than available, the length of the place list is implementation defined.

TABLE 21.1: Predefined Abstract Names for **OMP_PLACES**

Abstract Name	Meaning
threads	Each place corresponds to a single hardware thread on the device.
cores	Each place corresponds to a single core (having one or more hardware threads) on the device.
ll_caches	Each place corresponds to a set of cores that share the last level cache on the device.
numa_domains	Each place corresponds to a set of cores for which their closest memory on the device is: • the same memory; and • at a similar distance from the cores.
sockets	Each place corresponds to a single socket (consisting of one or more cores) on the device.

The behavior of the program is implementation defined when the execution environment cannot map a numerical value (either explicitly defined or implicitly derived from an interval) within the **OMP_PLACES** list to a processor on the target platform, or if it maps to an unavailable processor. The behavior is also implementation defined when the **OMP_PLACES** environment variable is defined using an abstract name.

The following grammar describes the values accepted for the **OMP_PLACES** environment variable.

⟨list⟩ ⊨ ⟨p-list⟩ | ⟨aname⟩
⟨p-list⟩ ⊨ ⟨p-interval⟩ | ⟨p-list⟩,⟨p-interval⟩
⟨p-interval⟩ ⊨ ⟨place⟩:⟨len⟩:⟨stride⟩ | ⟨place⟩:⟨len⟩ | ⟨place⟩ | !⟨place⟩
⟨place⟩ ⊨ {⟨res-list⟩} | ⟨res⟩
⟨res-list⟩ ⊨ ⟨res-interval⟩ | ⟨res-list⟩,⟨res-interval⟩
⟨res-interval⟩ ⊨ ⟨res⟩:⟨num-places⟩:⟨stride⟩ | ⟨res⟩:⟨num-places⟩ | ⟨res⟩ | !⟨res⟩
⟨aname⟩ ⊨ ⟨word⟩(⟨num-places⟩) | ⟨word⟩
⟨word⟩ ⊨ sockets | cores | ll_caches | numa_domains | threads | <implementation-defined abstract name>
⟨res⟩ ⊨ *non-negative integer*
⟨num-places⟩ ⊨ *positive integer*
⟨stride⟩ ⊨ *integer*
⟨len⟩ ⊨ *positive integer*

Examples:

```
setenv OMP_PLACES threads
setenv OMP_PLACES "threads(4)"
setenv OMP_PLACES
    "{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"
setenv OMP_PLACES "{0:4},{4:4},{8:4},{12:4}"
setenv OMP_PLACES "{0:4}:4:4"
```

where each of the last three definitions corresponds to the same 4 places including the smallest units of execution exposed by the execution environment numbered, in turn, 0 to 3, 4 to 7, 8 to 11, and 12 to 15.

Cross References

- *place-partition-var*, see Section 2.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **omp_get_num_places** routine, see Section 18.3.2.
- **omp_get_partition_num_places** routine, see Section 18.3.6.
- **omp_get_partition_place_nums** routine, see Section 18.3.7.
- **omp_get_place_num** routine, see Section 18.3.5.
- **omp_get_place_num_procs** routine, see Section 18.3.3.
- **omp_get_place_proc_ids** routine, see Section 18.3.4.

21.1.7 `OMP_PROC_BIND`

The **`OMP_PROC_BIND`** environment variable sets the initial value of the *bind-var* ICV. The value of this environment variable is either **`true`**, **`false`**, or a comma separated list of **`primary`**, **`master`** (**`master`** has been deprecated), **`close`**, or **`spread`**. The values of the list set the thread affinity policy to be used for parallel regions at the corresponding nested level.

If the environment variable is set to **`false`**, the execution environment may move OpenMP threads between OpenMP places, thread affinity is disabled, and **`proc_bind`** clauses on **`parallel`** constructs are ignored.

Otherwise, the execution environment should not move OpenMP threads between OpenMP places, thread affinity is enabled, and the initial thread is bound to the first place in the *place-partition-var* ICV prior to the first active parallel region. An initial thread that is created by a **`teams`** construct is bound to the first place in its *place-partition-var* ICV before it begins execution of the associated structured block.

If the environment variable is set to **`true`**, the thread affinity policy is implementation defined but must conform to the previous paragraph. The behavior of the program is implementation defined if the value in the **`OMP_PROC_BIND`** environment variable is not **`true`**, **`false`**, or a comma separated list of **`primary`**, **`master`** (**`master`** has been deprecated), **`close`**, or **`spread`**. The behavior is also implementation defined if an initial thread cannot be bound to the first place in the *place-partition-var* ICV.

The **`OMP_PROC_BIND`** environment variable sets the *max-active-levels-var* ICV to the number of active levels of parallelism that the implementation supports if the **`OMP_PROC_BIND`** environment variable is set to a comma-separated list of more than one element. The value of the *max-active-level-var* ICV may be overridden by setting **`OMP_MAX_ACTIVE_LEVELS`** or **`OMP_NESTED`**. See Section 21.1.4 and Section 21.1.5 for details.

Examples:

```
setenv OMP_PROC_BIND false
setenv OMP_PROC_BIND "spread, spread, close"
```

Cross References

- *bind-var* ICV, see Section 2.
- **`proc_bind`** clause, see Section 10.1.3.
- **`omp_get_proc_bind`** routine, see Section 18.3.1.

21.2 Program Execution Environment Variables

This section defines environment variables that affect program execution.

21.2.1 OMP_SCHEDULE

The **OMP_SCHEDULE** environment variable controls the schedule kind and chunk size of all loop directives that have the schedule kind **runtime**, by setting the value of the *run-sched-var* ICV.

The value of this environment variable takes the form:

[modifier:]kind[, chunk]

where

- *modifier* is one of **monotonic** or **nonmonotonic**;
- *kind* is one of **static**, **dynamic**, **guided**, or **auto**;
- *chunk* is an optional positive integer that specifies the chunk size.

If the *modifier* is not present, the *modifier* is set to **monotonic** if *kind* is **static**; for any other *kind* it is set to **nonmonotonic**.

If *chunk* is present, white space may be on either side of the "**,**". See Section 11.5 for a detailed description of the schedule kinds.

The behavior of the program is implementation defined if the value of **OMP_SCHEDULE** does not conform to the above format.

Examples:

```
setenv OMP_SCHEDULE "guided,4"
setenv OMP_SCHEDULE "dynamic"
setenv OMP_SCHEDULE "nonmonotonic:dynamic,4"
```

Cross References

- *run-sched-var* ICV, see Section 2.
- Worksharing-Loop construct, see Section 11.5.
- **omp_get_schedule** routine, see Section 18.2.12.
- **omp_set_schedule** routine, see Section 18.2.11.

21.2.2 OMP_STACKSIZE

The **OMP_STACKSIZE** environment variable controls the size of the stack for threads created by the OpenMP implementation, by setting the value of the *stacksize-var* ICV. The environment variable does not control the size of the stack for an initial thread.

The value of this environment variable takes the form:

size | *size***B** | *size***K** | *size***M** | *size***G**

where:

- *size* is a positive integer that specifies the size of the stack for threads that are created by the OpenMP implementation.
- **B**, **K**, **M**, and **G** are letters that specify whether the given size is in Bytes, Kilobytes (1024 Bytes), Megabytes (1024 Kilobytes), or Gigabytes (1024 Megabytes), respectively. If one of these letters is present, white space may occur between *size* and the letter.

If only *size* is specified and none of **B**, **K**, **M**, or **G** is specified, then *size* is assumed to be in Kilobytes.

The behavior of the program is implementation defined if **OMP_STACKSIZE** does not conform to the above format, or if the implementation cannot provide a stack with the requested size.

Examples:

```
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k "
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
setenv OMP_STACKSIZE "20 m "
setenv OMP_STACKSIZE " 1G"
setenv OMP_STACKSIZE 20000
```

Cross References

- *stacksize-var* ICV, see Section 2.

21.2.3 OMP_WAIT_POLICY

The **OMP_WAIT_POLICY** environment variable provides a hint to an OpenMP implementation about the desired behavior of waiting threads by setting the *wait-policy-var* ICV. A compliant OpenMP implementation may or may not abide by the setting of the environment variable.

The value of this environment variable must be one of the following:

active | **passive**

The **active** value specifies that waiting threads should mostly be active, consuming processor cycles, while waiting. An OpenMP implementation may, for example, make waiting threads spin.

The **passive** value specifies that waiting threads should mostly be passive, not consuming processor cycles, while waiting. For example, an OpenMP implementation may make waiting threads yield the processor to other threads or go to sleep.

The details of the **active** and **passive** behaviors are implementation defined.

The behavior of the program is implementation defined if the value of **OMP_WAIT_POLICY** is neither **active** nor **passive**.

Examples:

```
setenv OMP_WAIT_POLICY ACTIVE
setenv OMP_WAIT_POLICY active
setenv OMP_WAIT_POLICY PASSIVE
setenv OMP_WAIT_POLICY passive
```

Cross References

- *wait-policy-var* ICV, see Section 2.

21.2.4 OMP_DISPLAY_AFFINITY

The **OMP_DISPLAY_AFFINITY** environment variable instructs the runtime to display formatted affinity information by setting the *display-affinity-var* ICV. Affinity information is printed for all OpenMP threads in the parallel region upon entering the first parallel region and when any change occurs in the information accessible by the format specifiers listed in Table 21.2. If affinity of any thread in a parallel region changes then thread affinity information for all threads in that region is displayed. If the thread affinity for each respective parallel region at each nesting level has already been displayed and the thread affinity has not changed, then the information is not displayed again. Thread affinity information for threads in the same parallel region may be displayed in any order.

The value of the **OMP_DISPLAY_AFFINITY** environment variable may be set to one of these values:

true | false

The **true** value instructs the runtime to display the OpenMP thread affinity information, and uses the format setting defined in the *affinity-format-var* ICV.

The runtime does not display the OpenMP thread affinity information when the value of the **OMP_DISPLAY_AFFINITY** environment variable is **false** or undefined. For all values of the environment variable other than **true** or **false**, the display action is implementation defined.

Example:

```
setenv OMP_DISPLAY_AFFINITY TRUE
```

The above example causes an OpenMP implementation to display OpenMP thread affinity information during execution of the program, in a format given by the *affinity-format-var* ICV. The following is a sample output:

```
nesting_level=   1,   thread_num=   0,   thread_affinity=    0,1
nesting_level=   1,   thread_num=   1,   thread_affinity=    2,3
```

Cross References

- **OMP_AFFINITY_FORMAT** environment variable, see Section 21.2.5.
- Controlling OpenMP thread affinity, see Section 10.1.3.
- **omp_capture_affinity** routine, see Section 18.3.11.
- **omp_display_affinity** routine, see Section 18.3.10.
- **omp_get_affinity_format** routine, see Section 18.3.9.
- **omp_set_affinity_format** routine, see Section 18.3.8.

21.2.5 OMP_AFFINITY_FORMAT

The **OMP_AFFINITY_FORMAT** environment variable sets the initial value of the *affinity-format-var* ICV which defines the format when displaying OpenMP thread affinity information.

The value of this environment variable is case sensitive and leading and trailing whitespace is significant.

The value of this environment variable is a character string that may contain as substrings one or more field specifiers, in addition to other characters. The format of each field specifier is

`%`*[[*`0`*]*`.`*] size] type*

where an individual field specifier must contain the percent symbol (`%`) and a type. The type can be a single character short name or its corresponding long name delimited with curly braces, such as **%n** or **%{thread_num}**. A literal percent is specified as **%%**. Field specifiers can be provided in any order.

The **0** modifier indicates whether or not to add leading zeros to the output, following any indication of sign or base. The **.** modifier indicates the output should be right justified when *size* is specified. By default, output is left justified. The minimum field length is *size*, which is a decimal digit string with a non-zero first digit. If no *size* is specified, the actual length needed to print the field will be used. If the **0** modifier is used with *type* of **A**, **{thread_affinity}**, **H**, **{host}**, or a type that is not printed as a number, the result is unspecified. Any other characters in the format string that are not part of a field specifier will be included literally in the output.

TABLE 21.2: Available Field Types for Formatting OpenMP Thread Affinity Information

Short Name	Long Name	Meaning
t	**team_num**	The value returned by **omp_get_team_num()**.
T	**num_teams**	The value returned by **omp_get_num_teams()**.
L	**nesting_level**	The value returned by **omp_get_level()**.
n	**thread_num**	The value returned by **omp_get_thread_num()**.
N	**num_threads**	The value returned by **omp_get_num_threads()**.
a	**ancestor_tnum**	The value returned by **omp_get_ancestor_thread_num(***level***)**, where *level* is **omp_get_level()** minus 1.
H	**host**	The name for the host device on which the OpenMP program is running.
P	**process_id**	The process identifier used by the implementation.
i	**native_thread_id**	The native thread identifier used by the implementation.
A	**thread_affinity**	The list of numerical identifiers, in the format of a comma-separated list of integers or integer ranges, that represent processors on which a thread may execute, subject to OpenMP thread affinity control and/or other external affinity mechanisms.

Implementations may define additional field types. If an implementation does not have information for a field type, "undefined" is printed for this field when displaying the OpenMP thread affinity information.

Example:

```
setenv OMP_AFFINITY_FORMAT
       "Thread Affinity: %0.3L %.8n %.15{thread_affinity} %.12H"
```

The above example causes an OpenMP implementation to display OpenMP thread affinity information in the following form:

```
Thread Affinity: 001        0       0-1,16-17       nid003
Thread Affinity: 001        1       2-3,18-19       nid003
```

Cross References

- **OMP_DISPLAY_AFFINITY** environment variable, see Section 21.2.4.
- Controlling OpenMP thread affinity, see Section 10.1.3.

- **omp_capture_affinity** routine, see Section 18.3.11.
- **omp_display_affinity** routine, see Section 18.3.10.
- **omp_get_affinity_format** routine, see Section 18.3.9.
- **omp_set_affinity_format** routine, see Section 18.3.8.

21.2.6 OMP_CANCELLATION

The **OMP_CANCELLATION** environment variable sets the initial value of the *cancel-var* ICV.

The value of this environment variable must be one of the following:

true|**false**

If the environment variable is set to **true**, the effects of the **cancel** construct and of cancellation points are enabled and cancellation is activated. If the environment variable is set to **false**, cancellation is disabled and the **cancel** construct and cancellation points are effectively ignored. The behavior of the program is implementation defined if **OMP_CANCELLATION** is set to neither **true** nor **false**.

Cross References

- *cancel-var*, see Section 2.1.
- **cancel** construct, see Section 16.1.
- **cancellation point** construct, see Section 16.2.
- **omp_get_cancellation** routine, see Section 18.2.8.

21.2.7 OMP_DEFAULT_DEVICE

The **OMP_DEFAULT_DEVICE** environment variable sets the device number to use in device constructs by setting the initial value of the *default-device-var* ICV.

The value of this environment variable must be a non-negative integer value.

Cross References

- device directives, Section 13.
- *default-device-var* ICV, see Section 2.

21.2.8 OMP_TARGET_OFFLOAD

The **OMP_TARGET_OFFLOAD** environment variable sets the initial value of the *target-offload-var* ICV. The value of the **OMP_TARGET_OFFLOAD** environment variable must be one of the following:

mandatory | disabled | default

The **mandatory** value specifies that the effect of any device construct or device memory routine that uses a device that is unavailable or not supported by the implementation, or uses a non-conforming device number, is as if the **omp_invalid_device** device number was used.

Support for the **disabled** value is implementation defined. If an implementation supports it, the behavior is as if the only device is the host device.

The **default** value specifies the default behavior as described in Section 1.3.

Example:

```
% setenv OMP_TARGET_OFFLOAD mandatory
```

Cross References

- Device Directives, see Section 13.
- Device Memory Routines, see Section 18.8.
- *target-offload-var* ICV, see Section 2.

21.2.9 OMP_MAX_TASK_PRIORITY

The **OMP_MAX_TASK_PRIORITY** environment variable controls the use of task priorities by setting the initial value of the *max-task-priority-var* ICV. The value of this environment variable must be a non-negative integer.

Example:

```
% setenv OMP_MAX_TASK_PRIORITY 20
```

Cross References

- *max-task-priority-var* ICV, see Section 2.
- Tasking Constructs, see Section 12.
- **omp_get_max_task_priority** routine, see Section 18.5.1.

21.3 OMPT Environment Variables

This section defines environment variables that affect operation of the OMPT tool interface.

21.3.1 OMP_TOOL

The **OMP_TOOL** environment variable sets the *tool-var* ICV, which controls whether an OpenMP runtime will try to register a first party tool.

The value of this environment variable must be one of the following:

enabled | disabled

If **OMP_TOOL** is set to any value other than **enabled** or **disabled**, the behavior is unspecified. If **OMP_TOOL** is not defined, the default value for *tool-var* is **enabled**.

Example:

```
% setenv OMP_TOOL enabled
```

Cross References

- OMPT Interface, see Chapter 19.
- *tool-var* ICV, see Section 2.

21.3.2 OMP_TOOL_LIBRARIES

The **OMP_TOOL_LIBRARIES** environment variable sets the *tool-libraries-var* ICV to a list of tool libraries that are considered for use on a device on which an OpenMP implementation is being initialized. The value of this environment variable must be a list of names of dynamically-loadable libraries, separated by an implementation specific, platform typical separator. Whether the value of this environment variable is case sensitive is implementation defined.

If the *tool-var* ICV is not enabled, the value of *tool-libraries-var* is ignored. Otherwise, if **ompt_start_tool** is not visible in the address space on a device where OpenMP is being initialized or if **ompt_start_tool** returns *NULL*, an OpenMP implementation will consider libraries in the *tool-libraries-var* list in a left to right order. The OpenMP implementation will search the list for a library that meets two criteria: it can be dynamically loaded on the current device and it defines the symbol **ompt_start_tool**. If an OpenMP implementation finds a suitable library, no further libraries in the list will be considered.

Example:

```
% setenv OMP_TOOL_LIBRARIES libtoolXY64.so:/usr/local/lib/
    libtoolXY32.so
```

Cross References

- OMPT Interface, see Chapter 19.
- *tool-libraries-var* ICV, see Section 2.
- **ompt_start_tool** routine, see Section 19.2.1.

21.3.3 OMP_TOOL_VERBOSE_INIT

The **OMP_TOOL_VERBOSE_INIT** environment variable sets the *tool-verbose-init-var* ICV, which controls whether an OpenMP implementation will verbosely log the registration of a tool.

The value of this environment variable must be one of the following:

disabled | stdout | stderr | <filename>

If **OMP_TOOL_VERBOSE_INIT** is set to any value other than case insensitive **disabled**, **stdout** or **stderr**, the value is interpreted as a filename and the OpenMP runtime will try to log to a file with prefix *filename*. If the value is interpreted as a filename, whether it is case sensitive is implementation defined. If opening the logfile fails, the output will be redirected to stderr. If **OMP_TOOL_VERBOSE_INIT** is not defined, the default value for *tool-verbose-init-var* is **disabled**. Support for logging to **stdout** or **stderr** is implementation defined. Unless *tool-verbose-init-var* is **disabled**, the OpenMP runtime will log the steps of the tool activation process defined in Section 19.2.2 to a file with a name that is constructed using the provided filename prefix. The format and detail of the log is implementation defined. At a minimum, the log will contain the following:

- either that tool-var is disabled, or
- an indication that a tool was available in the address space at program launch, or
- the path name of each tool in **OMP_TOOL_LIBRARIES** that is considered for dynamic loading, whether dynamic loading was successful, and whether the **ompt_start_tool** function is found in the loaded library.

In addition, if an **ompt_start_tool** function is called the log will indicate whether or not the tool will use the OMPT interface.

Example:

```
% setenv OMP_TOOL_VERBOSE_INIT disabled
% setenv OMP_TOOL_VERBOSE_INIT STDERR
% setenv OMP_TOOL_VERBOSE_INIT ompt_load.log
```

Cross References

- OMPT Interface, see Chapter 19.
- *tool-verbose-init-var* ICV, see Section 2.

21.4 OMPD Environment Variables

This section defines environment variables that affect operation of the OMPD tool interface.

21.4.1 OMP_DEBUG

The **OMP_DEBUG** environment variable sets the *debug-var* ICV, which controls whether an OpenMP runtime collects information that an OMPD library may need to support a tool.

The value of this environment variable must be one of the following:

enabled | disabled

If **OMP_DEBUG** is set to any value other than **enabled** or **disabled** then the behavior is implementation defined.

Example:

```
% setenv OMP_DEBUG enabled
```

Cross References

- OMPD Interface, see Chapter 20.
- *debug-var* ICV, see Section 2.
- Enabling the Runtime for OMPD, see Section 20.2.1.

21.5 Memory Allocation Environment Variables

This section defines environment variables that affect memory allocations.

21.5.1 OMP_ALLOCATOR

The **OMP_ALLOCATOR** environment variable sets the initial value of the *def-allocator-var* ICV that specifies the default allocator for allocation calls, directives and clauses that do not specify an allocator.

The following grammar describes the values accepted for the **OMP_ALLOCATOR** environment variable.

$$\langle\text{allocator}\rangle \models \langle\text{predef-allocator}\rangle \mid \langle\text{predef-mem-space}\rangle \mid \langle\text{predef-mem-space}\rangle{:}\langle\text{traits}\rangle$$

$$\langle\text{traits}\rangle \models \langle\text{trait}\rangle{=}\langle\text{value}\rangle \mid \langle\text{trait}\rangle{=}\langle\text{value}\rangle{,}\langle\text{traits}\rangle$$

$$\langle\text{predef-allocator}\rangle \models \textit{one of the predefined allocators from Table 6.3}$$

$$\langle\text{predef-mem-space}\rangle \models \textit{one of the predefined memory spaces from Table 6.1}$$

$$\langle\text{trait}\rangle \models \textit{one of the allocator trait names from Table 6.2}$$

$$\langle\text{value}\rangle \models \textit{one of the allowed values from Table 6.2} \mid \textit{non-negative integer} \mid \langle\text{predef-allocator}\rangle$$

value can be an integer only if the *trait* accepts a numerical value, for the **fb_data** *trait* the *value* can only be *predef-allocator*. If the value of this environment variable is not a predefined allocator, then a new allocator with the given predefined memory space and optional traits is created and set as the *def-allocator-var* ICV. If the new allocator cannot be created, the *def-allocator-var* ICV will be set to **omp_default_mem_alloc**.

Example:

```
setenv OMP_ALLOCATOR omp_high_bw_mem_alloc
setenv OMP_ALLOCATOR omp_large_cap_mem_space:alignment=16,\
pinned=true
setenv OMP_ALLOCATOR omp_high_bw_mem_space:pool_size=1048576,\
fallback=allocator_fb,fb_data=omp_low_lat_mem_alloc
```

Cross References

- *def-allocator-var* ICV, see Section 2.
- Memory allocators, see Section 6.2.
- **omp_alloc** and **omp_aligned_alloc** routines, see Section 18.13.6
- **omp_calloc** and **omp_aligned_calloc** routines, see Section 18.13.8
- **omp_get_default_allocator** routine, see Section 18.13.5.
- **omp_set_default_allocator** routine, see Section 18.13.4.

21.6 Teams Environment Variables

This section defines environment variables that affect the operation of **teams** regions.

21.6.1 OMP_NUM_TEAMS

The **OMP_NUM_TEAMS** environment variable sets the maximum number of teams created by a **teams** construct by setting the *nteams-var* ICV.

The value of this environment variable must be a positive integer. The behavior of the program is implementation defined if the requested value of **OMP_NUM_TEAMS** is greater than the number of teams that an implementation can support, or if the value is not a positive integer.

Cross References

- *nteams-var* ICV, see Section 2.
- **omp_get_max_teams** routine, see Section 18.4.4.

21.6.2 OMP_TEAMS_THREAD_LIMIT

The **OMP_TEAMS_THREAD_LIMIT** environment variable sets the maximum number of OpenMP threads to use in each contention group created by a **teams** construct by setting the *teams-thread-limit-var* ICV.

The value of this environment variable must be a positive integer. The behavior of the program is implementation defined if the requested value of **OMP_TEAMS_THREAD_LIMIT** is greater than the number of threads that an implementation can support, or if the value is not a positive integer.

Cross References

- *teams-thread-limit-var* ICV, see Section 2.
- **omp_get_teams_thread_limit** routine, see Section 18.4.6.

21.7 OMP_DISPLAY_ENV

The **OMP_DISPLAY_ENV** environment variable instructs the runtime to display the information as described in the **omp_display_env** routine section (Section 18.15).

The value of the **OMP_DISPLAY_ENV** environment variable may be set to one of these values:

true | false | verbose

If the environment variable is set to **true**, the effect is as if the **omp_display_env** routine is called with the *verbose* argument set to *false* at the beginning of the program. If the environment variable is set to **verbose**, the effect is as if the **omp_display_env** routine is called with the *verbose* argument set to *true* at the beginning of the program. If the environment variable is undefined or set to **false**, the runtime does not display any information. For all values of the environment variable other than **true**, **false**, and **verbose**, the displayed information is unspecified.

Example:

```
% setenv OMP_DISPLAY_ENV true
```

For the output of the above example, see Section 18.15.

Cross References

- **omp_display_env** routine, see Section 18.15.

A OpenMP Implementation-Defined Behaviors

This appendix summarizes the behaviors that are described as implementation defined in this API. Each behavior is cross-referenced back to its description in the main specification. An implementation is required to define and to document its behavior in these cases.

Chapter 1:

- **Processor**: A hardware unit that is implementation defined (see Section 1.2.1).
- **Device**: An implementation defined logical execution engine (see Section 1.2.1).
- **Device pointer**: an implementation defined handle that refers to a device address (see Section 1.2.6).
- **Supported active levels of parallelism**: The maximum number of active parallel regions that may enclose any region of code in the program is implementation defined (see Section 1.2.7).
- **Memory model**: The minimum size at which a memory update may also read and write back adjacent variables that are part of another variable (as array elements or structure elements) is implementation defined but is no larger than required by the base language. The manner in which a program can obtain the referenced device address from a device pointer, outside the mechanisms specified by OpenMP, is implementation defined (see Section 1.4.1).

Chapter 2:

- **Internal control variables**: The initial values of *dyn-var*, *nthreads-var*, *run-sched-var*, *bind-var*, *stacksize-var*, *wait-policy-var*, *thread-limit-var*, *max-active-levels-var*, *place-partition-var*, *affinity-format-var*, *default-device-var*, *num-procs-var* and *def-allocator-var* are implementation defined (see Section 2.2).

Chapter 3:

C++

- Whether a **`throw`** executed inside a region that arises from an exception-aborting directive is treated as an **`error`** directive for which *sev-level* is **`fatal`** and *action-time* is **`execution`** is implementation defined.

C++

Chapter 4:

- **Canonical loop nest form**: The particular integer type used to compute the iteration count for the collapsed loop is implementation defined (see Section 4.4.1).

Chapter 5:

Fortran

- **Data-sharing attributes**: The data-sharing attributes of dummy arguments without the **VALUE** attribute are implementation defined if the associated actual argument is shared, except for the conditions specified (see Section 5.1.2).
- **threadprivate directive**: If the conditions for values of data in the threadprivate objects of threads (other than an initial thread) to persist between two consecutive active parallel regions do not all hold, the allocation status of an allocatable variable in the second region is implementation defined (see Section 5.2).

Fortran

Chapter 6:

- **Memory spaces**: The actual storage resources that each memory space defined in Table 6.1 represents are implementation defined (see Section 6.1).
- **Memory allocators**: The minimum partitioning size for partitioning of allocated memory over the storage resources is implementation defined. The default value for the **pool_size** allocator trait (see Table 6.2) is implementation defined. The associated memory space for each of the predefined **omp_cgroup_mem_alloc**, **omp_pteam_mem_alloc** and **omp_thread_mem_alloc** allocators (see Table 6.3) is implementation defined (see Section 6.2).

Chapter 7:

- **OpenMP context**: Whether the **dispatch** construct is added to the *construct* set, the accepted *isa-name* values for the *isa* trait, the accepted *arch-name* values for the *arch* trait, and the accepted *extension-name* values for the *extension* trait are implementation defined (see Section 7.1).
- **Metadirectives**: The number of times that each expression of the context selector of a **when** clause is evaluated is implementation defined (see Section 7.4).
- **Declare variant directive**: If two replacement candidates have the same score, their order is implementation defined. The number of times each expression of the context selector of a **match** clause is evaluated is implementation defined. For calls to **constexpr** base functions that are evaluated in constant expressions, whether any variant replacement occurs is implementation defined. Any differences that the specific OpenMP context requires in the prototype of the variant from the base function prototype are implementation defined (see Section 7.5).

- **`declare simd` directive**: If the parameter of the **`simdlen`** clause is not a constant positive integer expression, the number of concurrent arguments for the function is implementation defined. If the *alignment* parameter of the **`aligned`** clause is not specified, the default alignments for SIMD instructions are implementation defined (see Section 7.7).

- Whether the generated versions of a procedure that result from a declare target directive differ between devices or differ from the version of the procedure that is called from outside a **`target`** region is implementation defined (see Section 7.8).

Chapter 8:

- **`requires` directive**: Support for any feature specified by a requirement clause on a **`requires`** directive is implementation defined (see Section 8.2).

Chapter 9:

- **`unroll` construct**: If the **`partial`** clause is specified without an argument, the unroll factor is a positive integer that is implementation defined. If neither the **`partial`** nor the **`full`** clause is specified, if and how the loop is unrolled is implementation defined (see Section 9.2).

Chapter 10:

- **Dynamic adjustment of threads**: Providing the ability to adjust the number of threads dynamically is implementation defined (see Section 10.1.1).

- **Thread affinity**: For the **`close`** thread affinity policy, if $T > P$ and P does not divide T evenly, the exact number of threads in a particular place is implementation defined. For the **`spread`** thread affinity, if $T > P$ and P does not divide T evenly, the exact number of threads in a particular subpartition is implementation defined. The determination of whether the affinity request can be fulfilled is implementation defined. If not, the mapping of threads in the team to places is implementation defined (see Section 10.1.3).

- **`teams` construct**: The number of teams that are created is implementation defined, it is greater than or equal to the lower bound and less than or equal to the upper bound values of the **`num_teams`** clause if specified or it is less than or equal to the value of the *nteams-var* ICV if its value is greater than zero. Otherwise it is greater than or equal to 1. The maximum number of threads that participate in the contention group that each team initiates is implementation defined if no **`thread_limit`** clause is specified on the construct. The assignment of the initial threads to places and the values of the *place-partition-var* and *default-device-var* ICVs for each initial thread are implementation defined (see Section 10.2).

- **`simd` construct**: The number of iterations that are executed concurrently at any given time is implementation defined. If the *alignment* parameter is not specified in the **`aligned`** clause, the default alignments for the SIMD instructions are implementation defined (see Section 10.4).

Chapter 11:

- **single construct**: The method of choosing a thread to execute the structured block each time the team encounters the construct is implementation defined (see Section 11.1).
- **sections construct**: The method of scheduling the structured blocks among threads in the team is implementation defined (see Section 11.3).
- **Worksharing-loop directive**: The schedule that is used is implementation defined if the **schedule** clause is not specified. The effect of the **schedule(runtime)** clause when the *run-sched-var* ICV is set to **auto** is implementation defined. The value of *simd_width* for the **simd** schedule modifier is implementation defined (see Section 11.5).
- **distribute construct**: If no **dist_schedule** clause is specified then the schedule for the **distribute** construct is implementation defined (see Section 11.6).

Chapter 12:

- **taskloop construct**: The number of loop iterations assigned to a task created from a **taskloop** construct is implementation defined, unless the **grainsize** or **num_tasks** clause is specified (see Section 12.6).

C++

- **taskloop construct**: For **firstprivate** variables of class type, the number of invocations of copy constructors to perform the initialization is implementation defined (see Section 12.6).

C++

Chapter 13:

- **target construct**: The maximum number of threads that participate in the contention group that each team initiates is implementation defined if no **thread_limit** clause is specified on the construct (see Section 13.8).
- **is_device_ptr clause**: Support for pointers created outside of the OpenMP device data management routines is implementation defined (see Section 13.8).

Chapter 14:

- **interop directive**: The *foreign-runtime-id* that is used if the implementation does not support any of the items in *preference-list* is implementation defined (see Section 14.1).
- **interop Construct**: The *foreign-runtime-id* values for the **prefer_type** clause that the implementation supports, including non-standard names compatible with this clause, and the default choice when the implementation supports multiple values are implementation defined (see Section 14.1).
- The concrete types of the values of interop properties for implementation defined *foreign-runtime-ids* are implementation defined (see Section 14.1).

Chapter 15:

- **atomic construct**: A compliant implementation may enforce exclusive access between **atomic** regions that update different storage locations. The circumstances under which this occurs are implementation defined. If the storage location designated by *x* is not size-aligned (that is, if the byte alignment of *x* is not a multiple of the size of *x*), then the behavior of the atomic region is implementation defined (see Section 15.8.4).

Chapter 16:

- None.

Chapter 17:

- None.

Chapter 18:

C / C++

- **Runtime library definitions**: The enum types for **omp_allocator_handle_t**, **omp_event_handle_t**, **omp_interop_type_t** and **omp_memspace_handle_t** are implementation defined. The integral or pointer type for **omp_interop_t** is implementation defined (see Section 18.1).

C / C++

Fortran

- **Runtime library definitions**: Whether the include file **omp_lib.h** or the module **omp_lib** (or both) is provided is implementation defined. Whether the **omp_lib.h** file provides derived-type definitions or those routines that require an explicit interface is implementation defined. Whether any of the OpenMP runtime library routines that take an argument are extended with a generic interface so arguments of different **KIND** type can be accommodated is implementation defined (see Section 18.1).

Fortran

- **omp_set_num_threads routine**: If the argument is not a positive integer the behavior is implementation defined (see Section 18.2.1).
- **omp_set_schedule routine**: For implementation-specific schedule kinds, the values and associated meanings of the second argument are implementation defined (see Section 18.2.11).
- **omp_get_schedule routine**: The value returned by the second argument is implementation defined for any schedule kinds other than **static**, **dynamic** and **guided** (see Section 18.2.12).
- **omp_get_supported_active_levels routine**: The number of active levels of parallelism supported by the implementation is implementation defined, but must be greater than 0 (see Section 18.2.14).
- **omp_set_max_active_levels routine**: If the argument is not a non-negative integer then the behavior is implementation defined (see Section 18.2.15).

- **omp_get_place_proc_ids routine**: The meaning of the non-negative numerical identifiers returned by the **omp_get_place_proc_ids** routine is implementation defined. The order of the numerical identifiers returned in the array *ids* is implementation defined (see Section 18.3.4).

- **omp_set_affinity_format routine**: When called from within any **parallel** or **teams** region, the binding thread set (and binding region, if required) for the **omp_set_affinity_format** region and the effect of this routine are implementation defined (see Section 18.3.8).

- **omp_get_affinity_format routine**: When called from within any **parallel** or **teams** region, the binding thread set (and binding region, if required) for the **omp_get_affinity_format** region is implementation defined (see Section 18.3.9).

- **omp_display_affinity routine**: If the *format* argument does not conform to the specified format then the result is implementation defined (see Section 18.3.10).

- **omp_capture_affinity routine**: If the *format* argument does not conform to the specified format then the result is implementation defined (see Section 18.3.11).

- **omp_set_num_teams routine**: If the argument is not evaluated to a positive integer the behavior of this routine is implementation defined (see Section 18.4.3).

- **omp_set_teams_thread_limit routine**: If the argument is not a positive integer the behavior is implementation defined (see Section 18.4.5).

- **omp_target_memcpy_rect routine**: The maximum number of dimensions supported is implementation defined, but must be at least three (see Section 18.8.6).

- **Lock routines**: If a lock contains a synchronization hint, the effect of the hint is implementation defined (see Section 18.9 and Section 18.9.2).

Chapter 19:

- **ompt_callback_sync_region_wait**, **ompt_callback_mutex_released**, **ompt_callback_dependences**, **ompt_callback_task_dependence**, **ompt_callback_work**, **ompt_callback_master** (deprecated), **ompt_callback_masked**, **ompt_callback_target_map**, **ompt_callback_target_map_emi**, **ompt_callback_sync_region**, **ompt_callback_reduction**, **ompt_callback_lock_init**, **ompt_callback_lock_destroy**, **ompt_callback_mutex_acquire**, **ompt_callback_mutex_acquired**, **ompt_callback_nest_lock**, **ompt_callback_flush**, **ompt_callback_cancel** and **ompt_callback_dispatch tool callbacks**: If a tool attempts to register a callback with the string name using the runtime entry point **ompt_set_callback** (see Table 19.3), whether the registered callback may never, sometimes or always invoke this callback for the associated events is implementation defined (see Section 19.2.4).

- **Device tracing**: Whether a target device supports tracing or not is implementation defined; if a target device does not support tracing, a *NULL* may be supplied for the *lookup* function to the device initializer of a tool (see Section 19.2.5).
- **ompt_set_trace_ompt** and **ompt_buffer_get_record_ompt runtime entry points**: Whether a device-specific tracing interface will define this runtime entry point, indicating that it can collect traces in OMPT format is implementation defined. The kinds of trace records available for a device is implementation defined (see Section 19.2.5).
- **Native record abstract type**: The meaning of a *hwid* value for a device is implementation defined (see Section 19.4.3.3).
- **ompt_dispatch_chunk_t type**: Whether the chunk of a taskloop is contiguous is implementation defined (see Section 19.4.4.13).
- **ompt_record_abstract_t type**: The set of OMPT thread states supported is implementation defined (see Section 19.4.4.28).
- **ompt_callback_sync_region_t callback type**: For the *implicit-barrier-wait-begin* and *implicit-barrier-wait-end* event at the end of a parallel region, whether the **parallel_data** argument is *NULL* or points to the parallel data of the current parallel region is implementation defined (see Section 19.5.2.13).
- **ompt_callback_target_data_op_t callback type**: Whether in some operations *src_addr* or *dest_addr* might point to an intermediate buffer is implementation defined (see Section 19.5.2.25).
- **ompt_set_callback_t entry point type**: The subset of the associated event in which the callback is invoked is implementation defined (see Section 19.6.1.3).
- **ompt_get_place_proc_ids_t entry point type**: The meaning of the numerical identifiers returned is implementation defined. The order of *ids* returned in the array is implementation defined (see Section 19.6.1.8).
- **ompt_get_partition_place_nums_t entry point type**: The order of the identifiers returned in the array *place_nums* is implementation defined (see Section 19.6.1.10).
- **ompt_get_proc_id_t entry point type**: The meaning of the numerical identifier returned is implementation defined (see Section 19.6.1.11).

Chapter 20:

- **ompd_callback_print_string_fn_t callback function**: The value of *category* is implementation defined (see Section 20.4.5).
- **ompd_parallel_handle_compare operation**: The means by which parallel region handles are ordered is implementation defined (see Section 20.5.6.5).
- **ompd_task_handle_compare operation**: The means by which task handles are ordered is implementation defined (see Section 20.5.7.6).

Chapter 21:

- **OMP_DYNAMIC environment variable**: If the value is neither **true** nor **false** the behavior of the program is implementation defined (see Section 21.1.1).
- **OMP_NUM_THREADS environment variable**: If any value of the list specified leads to a number of threads that is greater than the implementation can support, or if any value is not a positive integer, then the behavior of the program is implementation defined (see Section 21.1.2).
- **OMP_THREAD_LIMIT environment variable**: If the requested value is greater than the number of threads an implementation can support, or if the value is not a positive integer, the behavior of the program is implementation defined (see Section 21.1.3).
- **OMP_MAX_ACTIVE_LEVELS environment variable**: If the value is not a non-negative integer or is greater than the maximum number of nested active parallel levels that an implementation can support then the behavior of the program is implementation defined (see Section 21.1.4).
- **OMP_NESTED environment variable (deprecated)**: If the value is neither **true** nor **false** the behavior of the program is implementation defined (see Section 21.1.5).
- **Conflicting OMP_NESTED (deprecated) and OMP_MAX_ACTIVE_LEVELS environment variables**: If both environment variables are set, the value of **OMP_NESTED** is **false**, and the value of **OMP_MAX_ACTIVE_LEVELS** is greater than 1, the behavior is implementation defined (see Section 21.1.5).
- **OMP_PLACES environment variable**: The meaning of the numbers specified in the environment variable and how the numbering is done are implementation defined. The precise definitions of the abstract names are implementation defined. An implementation may add implementation-defined abstract names as appropriate for the target platform. When creating a place list of n elements by appending the number *n* to an abstract name, the determination of which resources to include in the place list is implementation defined. When requesting more resources than available, the length of the place list is also implementation defined. The behavior of the program is implementation defined when the execution environment cannot map a numerical value (either explicitly defined or implicitly derived from an interval) within the **OMP_PLACES** list to a processor on the target platform, or if it maps to an unavailable processor. The behavior is also implementation defined when the **OMP_PLACES** environment variable is defined using an abstract name (see Section 21.1.6).
- **OMP_PROC_BIND environment variable**: If the value is not **true**, **false**, or a comma separated list of **primary** (**master** has been deprecated), **close**, or **spread**, the behavior is implementation defined. The behavior is also implementation defined if an initial thread cannot be bound to the first place in the OpenMP place list. The thread affinity policy is implementation defined if the value is **true** (see Section 21.1.7).
- **OMP_SCHEDULE environment variable**: If the value does not conform to the specified format then the behavior of the program is implementation defined (see Section 21.2.1).
- **OMP_STACKSIZE environment variable**: If the value does not conform to the specified format or the implementation cannot provide a stack of the specified size then the behavior is

implementation defined (see Section 21.2.2).

- **`OMP_WAIT_POLICY` environment variable**: The details of the **`active`** and **`passive`** behaviors are implementation defined (see Section 21.2.3).
- **`OMP_DISPLAY_AFFINITY` environment variable**: For all values of the environment variables other than **`true`** or **`false`**, the display action is implementation defined (see Section 21.2.4).
- **`OMP_AFFINITY_FORMAT` environment variable**: If the value does not conform to the specified format then the result is implementation defined (see Section 21.2.5).
- **`OMP_TARGET_OFFLOAD` environment variable**: The support of **`disabled`** is implementation defined (see Section 21.2.8).
- **`OMP_TOOL_LIBRARIES` environment variable**: Whether the value of the environment variable is case sensitive or insensitive is implementation defined (see Section 21.3.2).
- **`OMP_TOOL_VERBOSE_INIT` environment variable**: Support for logging to **`stdout`** or **`stderr`** is implementation defined. Whether the value of the environment variable is case sensitive when it is treated as a filename is implementation defined. The format and detail of the log is implementation defined (see Section 21.3.3).
- **`OMP_DEBUG` environment variable**: If the value is neither **`disabled`** nor **`enabled`** the behavior is implementation defined (see Section 21.4.1).
- **`OMP_NUM_TEAMS` environment variable**: If the value is not a positive integer or is greater than the number of teams that an implementation can support, the behavior of the program is implementation defined (see Section 21.6.1).
- **`OMP_TEAMS_THREAD_LIMIT` environment variable**: If the value is not a positive integer or is greater than the number of threads that an implementation can support, the behavior of the program is implementation defined (see Section 21.6.2).

B Features History

This appendix summarizes the major changes between OpenMP API versions since version 2.5.

B.1 Deprecated Features

The following features were deprecated in Version 5.2.

- The syntax of the **linear** clause that specifies its argument and *linear-modifier* as *linear-modifier* **(***list***)** was deprecated.
- The *minus* reduction was deprecated.
- For Fortran, the use of one or more **allocate** directives with an associated **ALLOCATE** statement was deprecated.
- The argument that specified the arguments of the **uses_allocators** clause as a comma-separated list in which each list item is a *clause-argument-specification* of the form *allocator[* **(***traits***)** *]* was deprecated.
- The use of the **default** clause on metadirectives was deprecated.

C / C++

- The delimited form of the **declare target** directive was deprecated.

C / C++

- The syntax of the **destroy** clause on the **depobj** construct with no argument was deprecated.
- The use of the keywords **source** and **sink** as *task-dependence-type* modifiers and the associated syntax for the **depend** clause was deprecated.
- The use of the **to** clause on the declare target directive was deprecated.

The following features were deprecated in Version 5.1.

- Cray pointer support was deprecated.
- The use of clauses supplied to the **requires** directive as context traits was deprecated.
- The **master** affinity policy was deprecated.
- The **master** construct and all combined and composite constructs of which it is a constituent construct were deprecated.

- The constant **omp_atv_sequential** was deprecated.
- In Fortran, specifying list items that are not of type **C_PTR** in a **use_device_ptr** or **is_device_ptr** clause was deprecated.
- The **ompt_sync_region_barrier** and **ompt_sync_region_barrier_implicit** values of the **ompt_sync_region_t** enum were deprecated.
- The **ompt_state_wait_barrier** and **ompt_state_wait_barrier_implicit** values of the **ompt_state_t** enum were deprecated.

The following features were deprecated in Version 5.0.

- The *nest-var* ICV, the **OMP_NESTED** environment variable, and the **omp_set_nested** and **omp_get_nested** routines were deprecated.
- Lock hints were renamed to synchronization hints. The following lock hint type and constants were deprecated:
  - the C/C++ type **omp_lock_hint_t** and the Fortran kind **omp_lock_hint_kind**;
  - the constants **omp_lock_hint_none**, **omp_lock_hint_uncontended**, **omp_lock_hint_contended**, **omp_lock_hint_nonspeculative**, and **omp_lock_hint_speculative**.

B.2 Version 5.1 to 5.2 Differences

- Numerous changes were made throughout the specification to improve quality of the specification of OpenMP syntax and to increase consistency of restrictions and their wording. These changes frequently result in the possible perception of differences to preceding versions of the OpenMP specification. However, those differences almost always resolve ambiguities, which may nonetheless have implications for existing implementations and programs.
- For OpenMP directives, reserved the **omp** sentinel (see Section 3.1, Section 3.1.2 and Section 3.1.1) and, for implementation-defined directives that extend the OpenMP directives reserved the **ompx** sentinel for C/C++ and free source form Fortran (see Section 3.1 and Section 3.1.2) and the **omx** sentinel for fixed source form Fortran to accomodate character position requirements (see Section 3.1.1). Reserved clause names that begin with the **ompx_** prefix for implementation-defined clauses on OpenMP directives (see Section 3.2). Reserved names in the base language that start with the **omp_** and **ompx_** prefix and reserved the **omp** and **ompx** namespaces (see Section 4) for the OpenMP runtime API and for implementation-defined extensions to that API (see Section 18).
- Allowed any clause that can be specified on a paired **end** directive to be specified on the directive (see Section 3.1), including the **copyprivate** clause (see Section 5.7.2) and the **nowait** clause in Fortran (see Section 15.6).

- For consistency with the syntax of other definitions of the clause, the syntax of the **destroy** clause on the **depobj** construct with no argument was deprecated (see Section 3.5).
- For consistency with the syntax of other clauses, the syntax of the **linear** clause that specifies its argument and *linear-modifier* as *linear-modifier* **(***list***)** was deprecated (see Section 5.4.6).
- The *minus* reduction operator was deprecated (see Section 5.5.6).
- To support the complete range of user-defined mappers and to improve consistency of **map** clause usage, the **declare mapper** directive was extended to accept *iterator-modifier* and the **present** *map-type-modifier* (see Section 5.8.2 and Section 5.8.10).
- The **enter** clause was added as a synonym for the **to** clause on the declare target directive, and the corresponding **to** clause was deprecated to reduce parsing ambiguity (see Section 5.10 and Section 7.8).

Fortran

- For consistency with other constructs with associated base language code, the executable form of the **allocate** directive and the **dispatch** construct were extended to allow an optional paired **end** directive to be specified (see Section 6.6 and Section 7.6).

Fortran

- The **allocators** construct was added to support the use of OpenMP allocators for variables that are allocated by a Fortran **ALLOCATE** statement, and the application of **allocate** directives to an **ALLOCATE** statement was deprecated (see Section 6.8).
- To support the full range of allocators and to improve consistency with the syntax of other clauses, the argument that specified the arguments of the **uses_allocators** as a comma-separated list in which each list item is a*clause-argument-specification* of the form *allocator[* **(***traits***)** *]* was deprecated (see Section 6.9).
- To improve code clarity and to reduce ambiguity in this specification, the **otherwise** clause was added as a synonym for the **default** clause on metadirectives and the corresponding **default** clause syntax was deprecated (see Section 7.4.2).

C / C++

- To improve overall syntax consistency and to reduce redundancy, the delimited form of the **declare target** directive was deprecated (see Section 7.8.2).

C / C++

- The **linear** clause was added to the syntax of the **distribute** construct to resolve an inconsistency between the syntax and description of the construct (see Section 11.6).
- To simplify usage, the **map** clause on a **target enter data** or **target exit data** construct now has a default map type that provides the same behavior as for the **to** or **from** map types, respectively (see Section 13.6 and Section 13.7).

- The **doacross** clause was added as a synonym for the **depend** clause with the keywords **source** and **sink** as *dependence-type* modifiers and the corresponding **depend** clause syntax was deprecated to improve code clarity and to reduce parsing ambiguity. Also, the **omp_cur_iteration** keyword was added to represent an iteration vector that refers to the current logical iteration (see Section 15.9.6).

B.3 Version 5.0 to 5.1 Differences

- Full support of C11, C++11, C++14, C++17, C++20 and Fortran 2008 was completed (see Section 1.7).
- Various changes throughout the specification were made to provide initial support of Fortran 2018 (see Section 1.7).
- To support device-specific ICV settings the environment variable syntax was extended to support device-specific variables (see Section 2.2 and Section 21).
- The OpenMP directive syntax was extended to include C++ attribute specifiers (see Section 3.1).
- The **omp_all_memory** reserved locator was added (see Section 3.1), and the **depend** clause was extended to allow its use (see Section 15.9.5).
- Support for **private** and **firstprivate** as an argument to the **default** clause in C and C++ was added (see Section 5.4.1).
- Support was added so that iterators may be defined and used in a motion clause in a **map** clause (see Section 5.8.2) or on a **target update** directive (see Section 13.9).
- The **present** argument was added to the **defaultmap** clause (see Section 5.8.9).
- Support for the **align** clause on the **allocate** directive and **allocator** and **align** modifiers on the **allocate** clause was added (see Section 6).
- The *target_device* trait set was added to the OpenMP Context (see Section 7.1), and the **target_device** selector set was added to context selectors (see Section 7.2).
- For C/C++, the declare variant directive was extended to support elision of preprocessed code and to allow enclosed function definitions to be interpreted as variant functions (see Section 7.5).
- The **declare variant** directive was extended with new clauses (**adjust_args** and **append_args**) that support adjustment of the interface between the original function and its variants (see Section 7.5).
- The **dispatch** construct was added to allow users to control when variant substitution happens and to define additional information that can be passed as arguments to the function variants (see Section 7.6).
- Support was added for indirect calls to the device version of a procedure or function in **target** regions. (see Section 7.8).

- Assumption directives were added to allow users to specify invariants (see Section 8.3).
- To support clarity in metadirectives, the **nothing** directive was added (see Section 8.4).
- To allow users to control the compilation process and runtime error actions, the **error** directive was added (see Section 8.5).
- Loop transformation constructs were added (see Section 9).
- The **masked** construct was added to support restricting execution to a specific thread (see Section 10.5).
- The **scope** directive was added to support reductions without requiring a **parallel** or worksharing region (see Section 11.2).
- The **grainsize** and **num_tasks** clauses for the **taskloop** construct were extended with a **strict** modifier to ensure a deterministic distribution of logical iterations to tasks (see Section 12.6).
- The **thread_limit** clause was added to the **target** construct to control the upper bound on the number of threads in the created contention group (see Section 13.8).
- The **has_device_addr** clause was added to the **target** construct to allow access to variables or array sections that already have a device address (see Section 13.8).
- The **interop** directive was added to enable portable interoperability with foreign execution contexts used to implement OpenMP (see Section 14.1). Runtime routines that facilitate use of **omp_interop_t** objects were also added (see Section 18.12).
- The **nowait** clause was added to the **taskwait** directive to support insertion of non-blocking join operations in a task dependence graph (see Section 15.5).
- Support was added for compare-and-swap and (for C and C++) minimum and maximum atomic operations through the **compare** clause. Support was also added for the specification of the memory order to apply to a failed comparing atomic operation with the **fail** clause (see Section 15.8.4).
- Specification of the **seq_cst** clause on a **flush** construct was allowed, with the same meaning as a **flush** construct without a list and without a clause (see Section 15.8.5).
- To support inout sets, the **inoutset** argument was added to the **depend** clause (see Section 15.9.5).
- The **omp_set_num_teams** and **omp_set_teams_thread_limit** runtime routines were added to control the number of teams and the size of those teams on the **teams** construct (see Section 18.4.3 and Section 18.4.5). Additionally, the **omp_get_max_teams** and **omp_get_teams_thread_limit** runtime routines were added to retrieve the values that will be used in the next **teams** construct (see Section 18.4.4 and Section 18.4.6).
- The **omp_target_is_accessible** runtime routine was added to test whether host memory is accessible from a given device (see Section 18.8.4).

- To support asynchronous device memory management, **omp_target_memcpy_async** and **omp_target_memcpy_rect_async** runtime routines were added (see Section 18.8.7 and Section 18.8.8).
- The **omp_get_mapped_ptr** runtime routine was added to support obtaining the device pointer that is associated with a host pointer for a given device (see Section 18.8.11).
- The **omp_calloc**, **omp_realloc**, **omp_aligned_alloc** and **omp_aligned_calloc** API routines were added (see Section 18.13).
- For the **omp_alloctrait_key_t** enum, the **omp_atv_serialized** value was added and the **omp_atv_default** value was changed (see Section 18.13.1).
- The **omp_display_env** runtime routine was added to provide information about ICVs and settings of environment variables (see Section 18.15).
- The **ompt_scope_beginend** value was added to the **ompt_scope_endpoint_t** enum to indicate the coincident beginning and end of a scope (see Section 19.4.4.11).
- The **ompt_sync_region_barrier_implicit_workshare**, **ompt_sync_region_barrier_implicit_parallel** and **ompt_sync_region_barrier_teams** values were added to the **ompt_sync_region_t** enum (see Section 19.4.4.14).
- Values for asynchronous data transfers were added to the **ompt_target_data_op_t** enum (see Section 19.4.4.15).
- The **ompt_state_wait_barrier_implementation** and **ompt_state_wait_barrier_teams** values were added to the **ompt_state_t** enum (see Section 19.4.4.28).
- The **ompt_callback_target_data_op_emi_t**, **ompt_callback_target_emi_t**, **ompt_callback_target_map_emi_t** and **ompt_callback_target_submit_emi_t** callbacks were added to support external monitoring interfaces (see Section 19.5.2.25, Section 19.5.2.26, Section 19.5.2.27 and Section 19.5.2.28).
- The **ompt_callback_error_t** type was added (see Section 19.5.2.30).
- The **OMP_PLACES** syntax was extended (see Section 21.1.6).
- The **OMP_NUM_TEAMS** and **OMP_TEAMS_THREAD_LIMIT** environment variables were added to control the number and size of teams on the **teams** construct (see Section 21.6.1 and Section 21.6.2).

B.4 Version 4.5 to 5.0 Differences

- The memory model was extended to distinguish different types of flush operations according to specified flush properties (see Section 1.4.4) and to define a happens before order based on synchronizing flush operations (see Section 1.4.5).
- Various changes throughout the specification were made to provide initial support of C11, C++11, C++14, C++17 and Fortran 2008 (see Section 1.7).
- Full support of Fortran 2003 was completed (see Section 1.7).
- The *target-offload-var* internal control variable (see Section 2) and the **OMP_TARGET_OFFLOAD** environment variable (see Section 21.2.8) were added to support runtime control of the execution of device constructs.
- Control over whether nested parallelism is enabled or disabled was integrated into the *max-active-levels-var* internal control variable (see Section 2.2), the default value of which is now implementation defined, unless determined according to the values of the **OMP_NUM_THREADS** (see Section 21.1.2) or **OMP_PROC_BIND** (see Section 21.1.7) environment variables.
- Support for array shaping (see Section 3.2.3) and for array sections with non-unit strides in C and C++ (see Section 3.2.4) was added to facilitate specification of discontiguous storage and the **target update** construct (see Section 13.9) and the **depend** clause (see Section 15.9.5) were extended to allow the use of shape-operators (see Section 3.2.3).
- Iterators (see Section 3.2.5) were added to support expressions in a list that expand to multiple expressions.
- The canonical loop form was defined for Fortran and, for all base languages, extended to permit non-rectangular loop nests (see Section 4.4.1).
- The *relational-op* in the *canonical loop form* for C/C++ was extended to include **!=** (see Section 4.4.1).
- To support conditional assignment to lastprivate variables, the **conditional** modifier was added to the **lastprivate** clause (see Section 5.4.5).
- The **inscan** modifier for the **reduction** clause (see Section 5.5.9) and the **scan** directive (see Section 5.6) were added to support inclusive and exclusive scan computations.
- To support task reductions, the **task** modifier was added to the **reduction** clause (see Section 5.5.9), the **task_reduction** clause (see Section 5.5.10) was added to the **taskgroup** construct (see Section 15.4), and the **in_reduction** clause (see Section 5.5.11) was added to the **task** (see Section 12.5) and **target** (see Section 13.8) constructs.
- To support taskloop reductions, the **reduction** (see Section 5.5.9) and **in_reduction** (see Section 5.5.11) clauses were added to the **taskloop** construct (see Section 12.6).

- The description of the **map** clause was modified to clarify the mapping order when multiple *map-types* are specified for a variable or structure members of a variable on the same construct. The **close** *map-type-modifier* was added as a hint for the runtime to allocate memory close to the target device (see Section 5.8.2).
- The capability to map C/C++ pointer variables and to assign the address of device memory that is mapped by an array section to them was added. Support for mapping of Fortran pointer and allocatable variables, including pointer and allocatable components of variables, was added (see Section 5.8.2).
- The **defaultmap** clause (see Section 5.8.9) was extended to allow selecting the data-mapping or data-sharing attributes for any of the scalar, aggregate, pointer or allocatable classes on a per-region basis. Additionally it accepts the **none** parameter to support the requirement that all variables referenced in the construct must be explicitly mapped or privatized.
- The **declare mapper** directive was added to support mapping of data types with direct and indirect members (see Section 5.8.10).
- Predefined memory spaces (see Section 6.1), predefined memory allocators and allocator traits (see Section 6.2) and directives, clauses and API routines (see Section 6 and Section 18.13) to use them were added to support different kinds of memories.
- The **metadirective** directive (see Section 7.4) and declare variant directive (see Section 7.5) were added to support selection of directive variants and declared function variants at a call site, respectively, based on compile-time traits of the enclosing context.
- Support for nested **declare target** directives was added (see Section 7.8).
- The **requires** directive (see Section 8.2) was added to support applications that require implementation-specific features.
- The **teams** construct (see Section 10.2) was extended to support execution on the host device without an enclosing **target** construct (see Section 13.8).
- The **loop** construct and the **order(concurrent)** clause were added to support compiler optimization and parallelization of loops for which iterations may execute in any order, including concurrently (see Section 10.3 and Section 11.7).
- The collapse of associated loops that are imperfectly nested loops was defined for the **simd** (see Section 10.4), worksharing-loop (see Section 11.5), **distribute** (see Section 11.6) and **taskloop** (see Section 12.6) constructs.
- The **simd** construct (see Section 10.4) was extended to accept the **if**, **nontemporal** and **order(concurrent)** clauses and to allow the use of **atomic** constructs within it.
- The default loop schedule modifier for worksharing-loop constructs without the **static** schedule and the **ordered** clause was changed to **nonmonotonic** (see Section 11.5).
- The **affinity** clause was added to the **task** construct (see Section 12.5) to support hints that indicate data affinity of explicit tasks.

- The **detach** clause for the **task** construct (see Section 12.5) and the **omp_fulfill_event** runtime routine (see Section 18.11.1) were added to support execution of detachable tasks.
- The **taskloop** construct (see Section 12.6) was added to the list of constructs that can be canceled by the **cancel** construct (see Section 16.1)).
- To support mutually exclusive inout sets, a **mutexinoutset** *dependence-type* was added to the **depend** clause (see Section 12.9 and Section 15.9.5).
- The semantics of the **use_device_ptr** clause for pointer variables was clarified and the **use_device_addr** clause for using the device address of non-pointer variables inside the **target data** construct was added (see Section 13.5).
- To support reverse offload, the **ancestor** modifier was added to the **device** clause for **target** constructs (see Section 13.8).
- To reduce programmer effort implicit declare target directives for some functions (C, C++, Fortran) and subroutines (Fortran) were added (see Section 13.8 and Section 7.8).
- The **target update** construct (see Section 13.9) was modified to allow array sections that specify discontiguous storage.
- The **to** and **from** clauses on the **target update** construct (see Section 13.9), the **depend** clause on task generating constructs (see Section 15.9.5), and the **map** clause (see Section 5.8.2) were extended to allow any lvalue expression as a list item for C/C++.
- Lock hints were renamed to synchronization hints, and the old names were deprecated (see Section 15.1).
- The **depend** clause was added to the **taskwait** construct (see Section 15.5).
- To support acquire and release semantics with weak memory ordering, the **acq_rel**, **acquire**, and **release** clauses were added to the **atomic** construct (see Section 15.8.4) and **flush** construct (see Section 15.8.5), and the memory ordering semantics of implicit flushes on various constructs and runtime routines were clarified (see Section 15.8.6).
- The **atomic** construct was extended with the **hint** clause (see Section 15.8.4).
- The **depend** clause (see Section 15.9.5) was extended to support iterators and to support depend objects that can be created with the new **depobj** construct.
- New combined constructs **master taskloop**, **parallel master**, **parallel master taskloop**, **master taskloop simd** **parallel master taskloop simd** (see Section 17.3) were added.
- The **omp_set_nested** (see Section 18.2.9) and **omp_get_nested** (see Section 18.2.10) routines and the **OMP_NESTED** environment variable (see Section 21.1.5) were deprecated.
- The **omp_get_supported_active_levels** routine was added to query the number of active levels of parallelism supported by the implementation (see Section 18.2.14).

- Runtime routines **omp_set_affinity_format** (see Section 18.3.8), **omp_get_affinity_format** (see Section 18.3.9), **omp_set_affinity** (see Section 18.3.10), and **omp_capture_affinity** (see Section 18.3.11) and environment variables **OMP_DISPLAY_AFFINITY** (see Section 21.2.4) and **OMP_AFFINITY_FORMAT** (see Section 21.2.5) were added to provide OpenMP runtime thread affinity information.
- The **omp_pause_resource** and **omp_pause_resource_all** runtime routines were added to allow the runtime to relinquish resources used by OpenMP (see Section 18.6.1 and Section 18.6.2).
- The **omp_get_device_num** runtime routine (see Section 18.7.5) was added to support determination of the device on which a thread is executing.
- Support for a first-party tool interface (see Section 19) was added.
- Support for a third-party tool interface (see Section 20) was added.
- Support for controlling offloading behavior with the **OMP_TARGET_OFFLOAD** environment variable was added (see Section 21.2.8).
- Stubs for Runtime Library Routines (previously Appendix A) were moved to a separate document.
- Interface Declarations (previously Appendix B) were moved to a separate document.

B.5 Version 4.0 to 4.5 Differences

- Support for several features of Fortran 2003 was added (see Section 1.7).
- The **if** clause was extended to take a *directive-name-modifier* that allows it to apply to combined constructs (see Section 3.4).
- The implicit data-sharing attribute for scalar variables in **target** regions was changed to **firstprivate** (see Section 5.1.1).
- Use of some C++ reference types was allowed in some data sharing attribute clauses (see Section 5.4).
- The **ref**, **val**, and **uval** modifiers were added to the **linear** clause (see Section 5.4.6).
- Semantics for reductions on C/C++ array sections were added and restrictions on the use of arrays and pointers in reductions were removed (see Section 5.5.9).
- Support was added to the map clauses to handle structure elements (see Section 5.8.2).
- To support unstructured data mapping for devices, the **map** clause (see Section 5.8.2) was updated and the **target enter data** (see Section 13.6) and **target exit data** (see Section 13.7) constructs were added.

- The **declare target** directive was extended to allow mapping of global variables to be deferred to specific device executions and to allow an *extended-list* to be specified in C/C++ (see Section 7.8).
- The **simdlen** clause was added to the **simd** construct (see Section 10.4) to support specification of the exact number of iterations desired per SIMD chunk.
- A parameter was added to the **ordered** clause of the worksharing-loop construct (see Section 11.5) and clauses were added to the **ordered** construct (see Section 15.9.7) to support doacross loop nests and use of the **simd** construct on loops with loop-carried backward dependences.
- The **linear** clause was added to the worksharing-loop construct (see Section 11.5).
- The **priority** clause was added to the **task** construct (see Section 12.5) to support hints that specify the relative execution priority of explicit tasks. The **omp_get_max_task_priority** routine was added to return the maximum supported priority value (see Section 18.5.1) and the **OMP_MAX_TASK_PRIORITY** environment variable was added to control the maximum priority value allowed (see Section 21.2.9).
- The **taskloop** construct (see Section 12.6) was added to support nestable parallel loops that create OpenMP tasks.
- To support interaction with native device implementations, the **use_device_ptr** clause was added to the **target data** construct (see Section 13.5) and the **is_device_ptr** clause was added to the **target** construct (see Section 13.8).
- The **nowait** and **depend** clauses were added to the **target** construct (see Section 13.8) to improve support for asynchronous execution of **target** regions.
- The **private**, **firstprivate** and **defaultmap** clauses were added to the **target** construct (see Section 13.8).
- The **hint** clause was added to the **critical** construct (see Section 15.2).
- The **source** and **sink** dependence types were added to the **depend** clause (see Section 15.9.5) to support doacross loop nests.
- To support a more complete set of device construct shortcuts, the **target parallel**, target parallel worksharing-loop target parallel worksharing-loop SIMD, and **target simd** (see Section 17.3) combined constructs were added.
- Query functions for OpenMP thread affinity were added (see Section 18.3.2 to Section 18.3.7).
- Device memory routines were added to allow explicit allocation, deallocation, memory transfers and memory associations (see Section 18.8).
- The lock API was extended with lock routines that support storing a hint with a lock to select a desired lock implementation for a lock's intended usage by the application code (see Section 18.9.2).

- C/C++ Grammar (previously Appendix B) was moved to a separate document.

B.6 Version 3.1 to 4.0 Differences

- Various changes throughout the specification were made to provide initial support of Fortran 2003 (see Section 1.7).
- C/C++ array syntax was extended to support array sections (see Section 3.2.4).
- The **reduction** clause (see Section 5.5.9) was extended and the **declare reduction** construct (see Section 5.5.12) was added to support user defined reductions.
- The **proc_bind** clause (see Section 10.1.3), the **OMP_PLACES** environment variable (see Section 21.1.6), and the **omp_get_proc_bind** runtime routine (see Section 18.3.1) were added to support thread affinity policies.
- SIMD directives were added to support SIMD parallelism (see Section 10.4).
- Implementation defined task scheduling points for untied tasks were removed (see Section 12.9).
- Device directives (see Section 13), the **OMP_DEFAULT_DEVICE** environment variable (see Section 21.2.7), and the **omp_set_default_device**, **omp_get_default_device**, **omp_get_num_devices**, **omp_get_num_teams**, **omp_get_team_num**, and **omp_is_initial_device** routines were added to support execution on devices.
- The **taskgroup** construct (see Section 15.4) was added to support more flexible deep task synchronization.
- The **atomic** construct (see Section 15.8.4) was extended to support atomic swap with the **capture** clause, to allow new atomic update and capture forms, and to support sequentially consistent atomic operations with a new **seq_cst** clause.
- The **depend** clause (see Section 15.9.5) was added to support task dependences.
- The **cancel** construct (see Section 16.1), the **cancellation point** construct (see Section 16.2), the **omp_get_cancellation** runtime routine (see Section 18.2.8) and the **OMP_CANCELLATION** environment variable (see Section 21.2.6) were added to support the concept of cancellation.
- The **OMP_DISPLAY_ENV** environment variable (see Section 21.7) was added to display the value of ICVs associated with the OpenMP environment variables.
- Examples (previously Appendix A) were moved to a separate document.

B.7 Version 3.0 to 3.1 Differences

- The *bind-var* ICV (see Section 2.1) and the **OMP_PROC_BIND** environment variable (see Section 21.1.7) were added to support control of whether threads are bound to processors.
- Data environment restrictions were changed to allow **intent(in)** and **const**-qualified types for the **firstprivate** clause (see Section 5.4.4).
- Data environment restrictions were changed to allow Fortran pointers in **firstprivate** (see Section 5.4.4) and **lastprivate** (see Section 5.4.5).
- New reduction operators **min** and **max** were added for C and C++ (see Section 5.5).
- The *nthreads-var* ICV was modified to be a list of the number of threads to use at each nested parallel region level and the algorithm for determining the number of threads used in a parallel region was modified to handle a list (see Section 10.1.1).
- The **final** and **mergeable** clauses (see Section 12.5) were added to the **task** construct to support optimization of task data environments.
- The **taskyield** construct (see Section 12.7) was added to allow user-defined task scheduling points.
- The **atomic** construct (see Section 15.8.4) was extended to include **read**, **write**, and **capture** forms, and an **update** clause was added to apply the already existing form of the **atomic** construct.
- The nesting restrictions in Section 17.1 were clarified to disallow closely-nested OpenMP regions within an **atomic** region so that an **atomic** region can be consistently defined with other OpenMP regions to include all code in the **atomic** construct.
- The **omp_in_final** runtime library routine (see Section 18.5.2) was added to support specialization of final task regions.
- Descriptions of examples (previously Appendix A) were expanded and clarified.
- Incorrect use of **omp_integer_kind** in Fortran interfaces was replaced with **selected_int_kind(8)**.

B.8 Version 2.5 to 3.0 Differences

- The definition of active **parallel** region was changed so that a **parallel** region is active if it is executed by a team that consists of more than one thread (see Section 1.2.2).
- The concept of tasks was added to the OpenMP execution model (see Section 1.2.5 and Section 1.3).
- The OpenMP memory model was extended to cover atomicity of memory accesses (see Section 1.4.1). The description of the behavior of **volatile** in terms of **flush** was removed.

- The definition of the *nest-var*, *dyn-var*, *nthreads-var* and *run-sched-var* internal control variables (ICVs) were modified to provide one copy of these ICVs per task instead of one copy for the whole program (see Section 2). The **omp_set_num_threads**, **omp_set_nested** and **omp_set_dynamic** runtime library routines were specified to support their use from inside a **parallel** region (see Section 18.2.1, Section 18.2.6 and Section 18.2.9).

- The *thread-limit-var* ICV, the **omp_get_thread_limit** runtime library routine and the **OMP_THREAD_LIMIT** environment variable were added to support control of the maximum number of threads that participate in the OpenMP program (see Section 2.1, Section 18.2.13 and Section 21.1.3).

- The *max-active-levels-var* ICV, the **omp_set_max_active_levels** and **omp_get_max_active_levels** runtime library routine and the **OMP_MAX_ACTIVE_LEVELS** environment variable and were added to support control of the number of nested active **parallel** regions (see Section 2.1, Section 18.2.15, Section 18.2.16 and Section 21.1.4).

- The *stacksize-var* ICV and the **OMP_STACKSIZE** environment variable were added to support control of the stack size for threads that the OpenMP implementation creates (see Section 2.1 and Section 21.2.2).

- The *wait-policy-var* ICV and the **OMP_WAIT_POLICY** environment variable were added to control the desired behavior of waiting threads (see Section 2.1 and Section 21.2.3).

- Predetermined data-sharing attributes were defined for Fortran assumed-size arrays (see Section 5.1.1).

- Static class members variables were allowed to appear in a **threadprivate** directive (see Section 5.2).

- Invocations of constructors and destructors for private and threadprivate class type variables was clarified (see Section 5.2, Section 5.4.3, Section 5.4.4, Section 5.7.1 and Section 5.7.2).

- The use of Fortran allocatable arrays was allowed in **private**, **firstprivate**, **lastprivate**, **reduction**, **copyin** and **copyprivate** clauses (see Section 5.2, Section 5.4.3, Section 5.4.4, Section 5.4.5, Section 5.5.9, Section 5.7.1 and Section 5.7.2).

- The **firstprivate** argument was added for the **default** clause in Fortran (see Section 5.4.1).

- Implementations were precluded from using the storage of the original list item to hold the new list item on the primary thread for list items in the **private** clause and the value was made well defined on exit from the **parallel** region if no attempt is made to reference the original list item inside the **parallel** region (see Section 5.4.3).

- Data environment restrictions were changed to allow **intent(in)** and **const**-qualified types for the **firstprivate** clause (see Section 5.4.4).

- Data environment restrictions were changed to allow Fortran pointers in **firstprivate** (see Section 5.4.4) and **lastprivate** (see Section 5.4.5).
- New reduction operators **min** and **max** were added for C and C++ (see Section 5.5).
- The rules for determining the number of threads used in a **parallel** region were modified (see Section 10.1.1).
- The assignment of iterations to threads in a loop construct with a **static** schedule kind was made deterministic (see Section 11.5).
- The worksharing-loop construct was extended to support association with more than one perfectly nested loop through the **collapse** clause (see Section 11.5).
- Iteration variables for worksharing-loops were allowed to be random access iterators or of unsigned integer type (see Section 11.5).
- The schedule kind **auto** was added to allow the implementation to choose any possible mapping of iterations in a loop construct to threads in the team (see Section 11.5).
- The **task** construct (see Section 12) was added to support explicit tasks.
- The **taskwait** construct (see Section 15.5) was added to support task synchronization.
- The runtime library routines **omp_set_schedule** and **omp_get_schedule** were added to set and to retrieve the value of the *run-sched-var* ICV (see Section 18.2.11 and Section 18.2.12).
- The **omp_get_level** runtime library routine was added to return the number of nested **parallel** regions that enclose the task that contains the call (see Section 18.2.17).
- The **omp_get_ancestor_thread_num** runtime library routine was added to return the thread number of the ancestor for a given nested level of the current thread, (see Section 18.2.18).
- The **omp_get_team_size** runtime library routine was added to return the size of the thread team to which the ancestor belongs for a given nested level of the current thread, (see Section 18.2.19).
- The **omp_get_active_level** runtime library routine was added to return the number of nested active **parallel** regions that enclose the task that contains the call (see Section 18.2.20).
- Lock ownership was defined in terms of tasks instead of threads (see Section 18.9).

Index

D