A Survey of High-Level Parallel Programming Models

Evgenij Belikov*, Pantazis Deligiannis, Prabhat Totoo,
Malak Aljabri, and Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, UK

Abstract

Increasingly heterogeneous and hierarchical parallel architectures are now mainstream, however, most of the traditional programming models are low-level and explicit, limiting portability, scalability, and productivity. Moreover, performance of applications that overspecify evaluation degree and order will suffer as they fail to adapt to changing architectures.

This paper surveys the properties, advantages, and disadvantages of high-level approaches to parallel programming that are deemed more flexible for efficiently utilising modern and future heterogeneous architectures. First, we introduce the challenges that arise from recent architectural trends and continue by surveying and characterising the high-level approaches to parallel programming. Subsequently, we review parallelism management policies and mechanisms to control them by comparing several representative implementations, focussing on heterogeneous hardware. Thereafter, we briefly illustrate the high-level approach based on recent case studies. Finally, we conclude by summarising our findings and views on what we believe are most suitable programming models to efficiently harness heterogeneous architectures without reducing productivity.

Keywords: High-Level Parallel Programming Models, Performance Portability, Parallel Patterns, Multi-Level Parallelism, Inherently Parallel Data Structures, Declarative Parallel Programming, Heterogeneous Computing

*Corresponding author: eb120@hw.ac.uk, School of Mathematical and Computer Sciences, Heriot-Watt University, EH14 4AS, Edinburgh, UK; +441314513432 (fax: -327)

1. Introduction

The multi-core revolution has brought the issue of parallel programming to the forefront of software development in general. Many parallel programming models have been developed in the past, however, there is no solidly established technology to pick up as the obvious choice for exploiting parallelism on today's mainstream multi-cores and on networks of multi-cores, serving as affordable high-performance platforms. Even more seriously, many of the established technologies are tied to one particular class of parallel machines, and are ill-suited to deal with radical architectural changes. To further boost computational capabilities, often graphics processing units (GPUs) or co-processors are added to this configuration, resulting in a highly heterogeneous computing platform. Programming such a platform is challenging for parallel computing experts, let alone domain experts new to parallel programming. The programming models for such a platform therefore need to strike a balance between simplifying the task of parallel programming while still delivering high performance on heterogeneous hardware, which is the motivation for using these platforms in the first place.

In this survey we aim to give guidance to *domain experts* in search of a high-level programming model that can exploit such heterogeneous hardware with minimal programmer effort. We explore the realm of parallel programming models, classify the established technologies on language as well as on system level, and assess their suitability for sustainable parallel programming in the long run, in particular envisioning further heterogeneity in the underlying platform, as discussed in Section 2. We identify several major challenges of parallel programming in Section 3: programmer productivity, performance portability, scalability, and adaptivity, which need to be addressed, since exploiting parallelism is the main source of increased performance.

We then explore several implementation technologies, dealing with these challenges. In particular, we classify the existing parallel programming models in Table 1 of Section 4 based on their respective level of abstraction over parallel coordination. In Section 5, we present a selection of low- and high-level models for programming heterogeneous architectures and discuss the current trends in GPU programming. Furthermore, we overview several representative policy control mechanisms in Table 2 of Section 6, and discuss the major system-level technologies that have proven successful across a range of parallel architectures. We summarise our findings from this in-depth study of models and systems in Section 8.

Throughout this survey we emphasise the importance of high-level approaches to parallel programming, as the most suitable basis for addressing architectural differences in heterogeneous machines and for facing further radical changes in hardware design, for example in the form of many-core co-processors. We believe that with the shift of parallel programming from *supercomputing parallelism*, performed by expert parallel programmers on specialised hardware with huge investment of time and effort, to *desktop parallelism*, performed by domain experts on off-the-shelf hardware with only minimal effort, such high-level programming models are the best technology to meet programmers' demands. Since they do not tie the parallel application to one particular architecture, they offer high potential in terms of scalability, when moving to clusters of multi-cores, and in terms of added heterogeneity, when adding GPUs or other non-standard computing engines to the configuration. Of course, this imposes further challenges on the underlying system that implements such a high-level parallel programming model.

We don't aim to give a comprehensive survey of implementation techniques, which would be much beyond the scope of this report, but we classify and discuss essential aspects of the existing implementation approaches of high-level models, and give references for further reading beyond our discussion. We separate the discussion in this survey into a section on programming models (Section 4) and underlying systems (Section 6). We specifically focus on applying high-level programming models to heterogeneous hardware in Section 5. While undoubtedly supercomputing parallelism will remain a strong research area, we don't survey this part of technology landscape, and focus our study on the area of desktop parallelism, presenting a comprehensive survey of literature in this field, hoping to guide computer scientists, who are now entering this area and are in search of a suitable technology today, that will remain relevant tomorrow.

2. Trends in Parallel Architectures

Hardware architecture substantially affects application performance [1]. Therefore, it is important to take the characteristics of the target architecture into account, especially as novel parallel architectures are increasingly heterogeneous and hierarchical. Most of them fit into single-instruction-multiple-data (SIMD) or multiple-instruction-multiple-data (MIMD) category of the well-known Flynn's taxonomy [2] that classifies architectures according to the number of data streams and instruction streams.

MIMD category is futher subdivided into shared-memory and distributed-memory architectures. Moreover, shared-memory architectures can be classified as either symmetric multiprocessors with uniform memory access (UMA) or as non-uniform memory access (NUMA) architectures that have recently gained popularity [3, 4]. Finally, the architectures can be distinguished based on differences in cache sharing, number of hierarchy levels, and degree of heterogeneity of the components. In this section, the focus is on the recent architectural trends that substantially complicate parallel programming.

2.1. Towards Many-cores

Since the end of the frequency scaling era [5], there is a clear trend towards *many-core architectures*. Whilst mobile phones readily have multiple cores, architectures such as Intel's TeraScale and Tilera's TILE-Gx100 use 80 and 100 cores respectively, whereas NVIDIA's GT300 GPU uses 512 scalar processors [6]. However, scalability of current shared-memory designs is limited by increasing the overhead of maintaining coherent view of shared caches [7], leading to non-cache-coherent and Network-on-Chip-based designs [8].

Furthermore, it is likely that *no one-fits-all architecture* will emerge in the near future, due to the trade-offs between raw performance, power-efficiency, manufacturing costs, and programmability. Additionally, different architectures appear particularly suitable for specific kinds of applications. For example, GPUs excel at regular data-parallel applications that use floating-point operations [9], whereas FPGAs are suitable for streaming applications that use integer or logic operations [10, 6], and clusters, GRIDs, or Clouds [11, 12, 13, 14] provide scalability beyond a single node being most suitable for large-scale and High-Performance Computing (HPC) applications. Although facing diminishing returns, *the number of cores is expected to continue increasing* in the foreseeable future.

2.2. Heterogeneity and Dark Silicon

Another trend in parallel architectures is to include cores of differing capabilities [15, 16, 17]. In contrast to homogeneous designs, heterogeneous ones are more versatile for general purpose computing, since they are more flexible in utilising the available transistor budget and power envelope [18]. Whilst larger CPUs are suitable for handling task parallel and sequential program sections, on-chip GPUs excel at regular data-parallelism [9, 19, 20].

Further heterogeneity results from the need to reduce power consumption and due to physical limitations that make it impossible to simultaneously

power-up all the available transistors, an effect referred to as Dark Silicon [7]. This effect leads to architectures that are capable of dynamically changing their characteristics during program execution. This diversity and rapid architectural evolution mandate automated and adaptive solutions.

2.3. Memory and Network Hierarchy

Parallel architectures are increasingly *hierarchical* regarding memory access and network interconnect. Multi-level caches are used to reduce the frequency of accesses to the relatively slow main memory to increase memory bandwidth and to hide access latency. In many modern architectures over 80% of the chip area is devoted to latency hiding.

By sharing the same cache, cores may communicate more efficiently, however pathological cases such as cache thrashing need to be avoided [21]. Moreover cache coherence protocols are not likely to scale to manycores due to increasing overheads, making non-cache-coherent architectures a viable alternative. Hence, although the instruction count was predominantly used in the past, currently, the amount and structure of memory accesses should be taken into account to exploit data locality for performance. Network hierarchy can be viewed as generalisation of the memory hierarchy, thus highlighting the impact of latency and bandwidth on performance of applications that rely on significant amount of communication [22]. Moreover, the actual topology of and congestion within the network may severely impact performance and hence should be taken into account.

All the aforementioned trends pose substantial challenges for parallel programming models and language design as they complicate efficient utilisation of the available resources. Attacking the challenges presented below is critical, since *exploiting parallelism emerges as the main source of further gains in application performance.*

3. Challenges in Exploiting Parallelism

Exploiting parallelism for performance is difficult due to well-known challenges that arise from the increasing number of processing elements (PEs) and from the need to coordinate access to shared resources, exacerbated by the heterogeneity of components and deep memory and network hierarchies.

3.1. The Complexity of Parallel Programming

Parallel programming, i.e. the translation of an algorithmic solution to a problem (the *what*) into a correct and efficient program that fully exploits the underlying parallel architecture, is substantially more difficult than sequential programming due to the added complexity of *coordination* aspects [23, 24], i.e. *how* the *computation* is decomposed, mapped to, communicated among, and synchronised across the available PEs [25, 26, 27].

Whilst exploitation of hardware-specific features remains crucial for performance [28, 29, 30], rapid architectural evolution, increasing software complexity, and large non-linear design space render manual code adaptation infeasible, due to reduced productivity and compromised portability. Hence, there is a need for a unified high-level parallel programming model that transparently adapts to a wide range of target architectures, achieving high *performance* without sacrificing high *productivity* and *portability*.

Coordination. Coordination deals with access to shared resources, parallelism management, and inter-process communication. The computation needs to be partitioned to enable parallel execution. Often, complex data structures need to be serialised by the sender and deserialised by the receiver. There is a trade-off between the communication overhead of balancing the load to increase resource utilisation and preserving data locality to reduce communication overhead. Thus, determining a good mapping of tasks to PEs poses yet another challenge. Alas, most coordination decisions are NP-hard, so it is difficult to obtain optimal solutions, and all coordination mechanisms contribute to the overhead which may cancel out the benefits of parallelism.

Unfortunately, many mainstream languages were not designed with parallelism in mind and are poorly suited for exploiting parallel architectures. For example, the model of *threads* that explicitly share memory is *non-deterministic* and is difficult to use to achieve deterministic goals, as the programmer has to employ such low-level mechanisms as locks to ensure mutually exclusive access to shared resources. Programming mistakes can lead to *race conditions* and non-composability makes using multiple locks prone to *deadlocks* or *livelocks* [31, 32, 33] — errors that are notoriously hard to detect and correct. Furthermore, using too coarse-grained locks reduces parallelism, whereas too fine-grained locks result in excessive overheads, both limiting scalability and overall performance.

Performance Portability. To fully exploit modern heterogeneous architectures, there is a need to achieve high *performance portability*, i.e. the ability

of software to maintain high performance across a wide range of parallel architectures. Rapid architectural evolution renders manual code adaptation infeasible and mandates adaptive solutions. To adapt, architectural and system information, such as the number of PEs, memory hierarchy, communication latency, and system load, should influence coordination decisions alongside the inferred sub-task size for a given application and input [34].

Productivity. Productivity gains are responsible for the popularity of structured, object-oriented, and declarative approaches to programming, since using higher-level languages requires less programmer effort to solve more complex problems.

The expected lifetime and a large user base of some systems justify the large development effort of using an error-prone low-level language that offers high degree of control for peak performance. However, this programming model requires intricate knowledge of the target architecture on par of the programmer and encourages non-portable optimisations. Non-determinism and the associated difficulty of verification and performance analysis due to exponentially larger behaviour space often prohibitively limit productivity.

Hence, since many programmers were trained using the sequential model of computation, a solution is desirable that handles parallelism as transparently as possible. Ideally, no specialist architectural knowledge would be required, since the infrastructure would implicitly manage parallelism. Although feasible for restricted parallel patterns or data structures, fully automatic parallelisation unfortunately proved intractable in general. This requires programmers to change their mindset to *think parallel*. Nevertheless, a high-level programming model seems to offer higher productivity an flexibility by automating a large fraction of lower-level decisions where possible.

Scalability. Scalability refers to the ability of a program to utilise increasing available resources, e.g. memory size and the number of PEs, to solve larger problems faster. Although some problems that comprise independent compute-bound tasks that require negligible volume of communication are *pleasingly parallel*, many problems have non-obvious upper bounds on resources they could efficiently utilise before the benefit of parallelisation is overwhelmed by the associated overheads in addition to the inherent limits imposed by the sequential fraction of the algorithm [35].

Fine-grained parallelism is potentially more scalable and flexible in adapting to changing conditions, since sub-tasks can be inlined if necessary or proceed in parallel, depending on the architecture and on the execution state [36].

In many cases, programs written in low-level languages suffer losses in performance if moved to an architecture for which the implicit assumptions made by the programmers no longer hold. For example, fixing granularity is likely to result either in load imbalance, if the tasks are too coarse, or in excessive overhead, if the tasks are too fine-grained. Both cases lead to substantial performance degradation and reduced scalability [37, 38].

3.2. Automated Management of Parallelism

Finding a system-driven and adaptive way of transparently mapping high-level language constructs to diverse target architectures in response to changing conditions is crucial for scalability and performance portability. This involves analysing the cost of parallel execution based on relevant parameters and dynamically controlling different policies.

Analysis of Parallel Execution Costs. Exploiting parallelism is worthwhile only if the cost of parallelisation and of parallel execution is lower than the benefit compared to the sequential version. Therefore, to dynamically decide whether a computation should be split up and run in parallel requires information on the cost of parallel computation and the cost of coordination.

Modelling these costs is non-trivial, since the cost information needs to be accurate despite noise and uncertainty, up-to-date, and reflect tightly the actual execution costs [34]. Many traditional models have a small set of parameters and are intended for use at design time [39, 40]. However, such models often neglect parameters that have substantial effect on performance, e.g. memory access and network latencies or system load.

Dynamic Policy Control. Based on the execution model and obtained information, policies can be dynamically controlled to improve the management of parallelism. Many mechanisms suggested in the literature are based on heuristics, which work excellently in certain situations, but fail in others. Thus, the promise of dynamic control is to detect a failing heuristic and to switch to a better one. Scalable decentralised mechanisms such as work-stealing [41], enhanced by incorporating architectural and system information to improve adaptation potential, appear particularly promising.

Dynamic adaptation requires *change detection* or inference based on a set of observed parameter values, which are then *analysed* and compared to the desired outcome, whereafter adaptation is *planned* and *executed*. This is similar to the cognition cycle used in self-managing systems [42, 43] and to feedback-directed control [44].

3.3. Multi-Level Parallelism

To exploit the increasingly hierarchical architectures, several hybrid programming models have been proposed (e.g. MPI + OpenMP + CUDA [45]) that use a different programming model at different hierarchy level to exploit shared-memory, distributed-memory, and massive data parallelism. However, these models are low-level and non-uniform prohibitively reducing productivity and performance portability [46, 47].

The OpenCL framework and OmpSs are geared towards unifying multi-level parallel programming across CPUs and GPUs [48, 49]. Unfortunately, the adopted model is rather low-level, leaving many coordination decisions to the programmer.

By contrast, more structured higher-level approaches shift many of the parallelism management responsibilities to the OS, the compiler, and the run-time system (RTS), thus increasing productivity and performance portability [50, 51]. The challenge is to integrate potentially nested data and task parallelism and to design *composable* language constructs.

3.4. Inherently Parallel Data Structures

Many traditional data structures such as linked lists have sequential representations that encourage sequential operations making it difficult to exploit parallelism by unnecessarily restricting the degree and order of evaluation. Sequential operations increase the serial fraction of the program, which according to Amdahl's Law reduces the upper bound on speedup [35]. Therefore, parallel data structures that are optimised to exploit parallelism, both in the representation and in the associated operations, are crucial for developing efficient parallel applications and systems [52]. Novel data structures need more flexibility to maintain high performance across diverse heterogeneous parallel architectures.

In multithreaded environments, concurrent data structures can help to avoid blocking in order to improve scalability, facilitate parallel programming by hiding low-level synchronisation details, and ensure *thread-safe* access to shared data. The main challenge is to devise operations on data structures that are implicitly parallel and hide the coordination issues from the programmer. To achieve this, careful selection of appropriate representation of data structures in memory is important. For instance, many algorithms can be rewritten to take advantage of parallel operations over a tree-like data representation which lends itself to parallelisation, instead of using the inherently sequential alternative [53].

4. High-Level Parallel Programming Models

Finding a clear-cut classification of parallel programming models, systems and languages is difficult since there are several meaningful ways of grouping them. In Figure 1, we present an overview of models across the dimensions of computation (i.e. the algorithmic solution) and coordination (the management of parallelism).

Table 1 provides a more detailed classification based on seven main classes, listed in the first column in ascending order of abstraction within each group. In each class, a representative number of languages is presented and for each language we report on some key properties regarding the type of parallelism, the memory model, determinism, and embedding of the constructs for parallelism into the host language.

4.1. Language Properties

The properties are listed across the columns and highlight key characteristics of a language.

4.1.1. Coordination Abstraction

The coordination abstraction refers to the degree of explicit control required by the programmer to manage parallelism and access to shared resources. Higher level of abstraction leads to higher productivity and reduced risk of introducing errors in the parallel program at the potential cost of decreased performance.

Low-Level Models. Such models, e.g. Java Threads or MPI, expose most coordination issues such as problem decomposition, communication, and synchronisation to the programmer [54]. These issues are orthogonal to the algorithmic problem, require additional effort, and thus reduce productivity. Dealing with these is notoriously difficult, which constitutes the challenge of parallel programming. Low-level models offer extensive tuning opportunities for expert programmers at the cost of significant effort.

Mid-Level Models. Models in this category hide some of the coordination issues from the programmer, in particular thread and memory management, and mapping of work units to threads. For instance, in OpenMP, the programmer uses directives to identify parallel regions and the compiler generates the threaded code.

The Task Parallel Library [55] can also be classified as mid-level. Although it provides task abstraction and the run-time system automatically maps tasks that represent distinct units of work to worker threads, the programmer is still responsible for synchronisation and splitting work into tasks.

Mid-level models attempt to strike a balance between the performance benefits through available tuning opportunities and the productivity advantage through the increased level of abstraction.

High-Level Models. These models abstract over most of the coordination, often leaving only advisory identification of parallelism to the programmer. Usually built on top of basic parallel constructs, e.g. Java threads or the par combinator in GpH, these models provide a more structured way of describing parallelism through the use of abstractions that encapsulate common patterns of computation and coordination.

For instance, algorithmic skeletons can offer an architecture-independent interface while providing multiple parallel back-ends to retain performance across different architectures [56]. The programmer needs merely to select suitable skeletons and get parallelism for free. High-level models offer the most powerful abstractions whilst substantially complicating the efficient implementation of the underlying language or library.

4.1.2. Parallelism Types

There are two common types of parallelism – *data parallelism* that refers to performing an operation on separate blocks of data in parallel; and *task parallelism* where parallel execution is structured based on inter-dependencies among separate tasks. Most of the models support both types of parallelism. Efficiently integrating data, nested data and task parallelism remains an open issue since implementing each type of parallelism seems to require different mechanisms.

4.1.3. Memory Model

The memory model describes how threads interact to exchange non-shared data and synchronise to coordinate the access to shared memory.

Shared Memory. Thread-based parallelism, such as pthreads [57] or Java Threads [58], replaced processes as a predominant programming model on shared-memory architectures. However, this explicit approach has proven too low-level to facilitate structured parallel programming due to the need to avoid race conditions, specify synchronisation and communication, and

prevent deadlocks [33]. Sharing memory often requires fine-grained locking resulting in scalability issues. Higher-level approaches such as OpenMP [59] were introduced that hide thread management from the programmers who use pragmas to demand parallel execution. However, explicit locking is required to prevent race conditions and the scope of variables needs to be manually specified.

Transactional Memory. Inspired by result from database research, Transactional memory (TM) [60, 61, 62] is another approach to synchronisation. TM allows instruction blocks called *transactions* to execute *atomically*, i.e. they either succeed or have no effects at all. This is achieved by optimistically executing the transaction whilst logging the accesses and rolling back to a consistent state if the transaction fails due to an access conflict. TM provides a composable high-level abstraction for shared-memory programming, however, empirical evidence regarding performance and scalability of software transactional memory is as yet inconclusive [63, 64].

Message Passing. To exploit distributed-memory architectures, the message-passing model has emerged as a standard approach to program applications, in particular for large-scale homogeneous and flat HPC architectures. Although low-level and unstructured, the model avoids most of the issues associated with the shared-memory model by not sharing any state at the cost of message-buffering and encoding overheads. Exposure of the low-level details, tempts the programmer to manually optimise the application for a given target architecture, often decreasing performance portability. The level of abstraction is raised by using collective operations [65] akin to algorithmic skeletons, and by delegating their efficient implementation to the library. The libraries such as PVM [66] and MPI [67] are likely to remain important as the lower communication layer of higher-level libraries and languages, where the challenge is in adapting to heterogeneous and hierarchical architectures. Some languages (e.g. Scala, Erlang) provide support for light-weight processes based on the Actor model [68, 69]. Despite improved scalability it is not clear whether message passing needs to remain explicit or can be efficiently handled by the RTS. Unlike the shared-memory model, the message-passing model can be used to program both shared-memory and distributed-memory architectures.

PGAS. The concept of Partitioned Global Address Space (PGAS) enables the programmer to use the abstraction of virtual shared-memory, while pro-

viding possibilities for co-locating data on specific nodes and thereby tune the parallel execution. PGAS is covered in more detail in Section 4.2.4 below.

4.1.4. Determinism

A deterministic model guarantees that the parallel execution yields the same results as the sequential execution of a program. For example, invoking the parallel version of a query on a PLINQ object and the par combinator in GpH, are both deterministic. Deterministic models can be achieved by design or implemented by building on top of non-deterministic constructs and providing a deterministic library, e.g Par monad. Deterministic programming models abstract over low-level thread management issues such as synchronisation, hence preventing the appearance of race conditions and deadlocks.

By contrast, programming models such as Java Threads that depend on basic concurrency constructs to implement parallelism are non-deterministic, requiring the programmer to explicitly manage the synchronisation among threads to ensure correct program behaviour.

4.1.5. Embedding

There is a range of different technologies to embed support for parallel programming into a host language. Starting from a fresh language design, first-class primitives for parallelism are the most obvious choice, maximising the flexibility and allowing to use standard language concepts. When extending an existing language, new parallel features can be provided as preprocessor or compiler directives, or built on top of available low-level concurrency primitives. Often libraries are used to provide similar features in a less invasive way [70, 71]. However, this approach is restricted to the optimisations available in the host language. Alternatively, a separate coordination language can be used to specify parallel execution and communication [72], separating the concerns of computation and coordination. Sufficiently high-level languages enable seamless embedding of the coordination language in the host language, as exemplified by Evaluation Strategies [73] for GpH.

4.2. Classes of Parallel Programming Models

We group different programming models based on the emerging clusters as depicted in Figure 1. Below we discuss each group drawing specific examples from Table 1.

Table 1: Parallel Programming Models, Languages and Systems

	Language/extension	Coordination abstraction	Type	Memory model	Deter-ministic	Embedding
Par.Impe.	MPI/PVM	low	task, data	msg pass (expl.)	no	library
	OpenMP	mid	data, task	shared (expl.)	no	compiler directives
	Cilk	mid	task, data	shared	no	C extension
	TBB	mid	data, task	shared	no	C++ library
Par. OO	Java Threads	low	task, data	shared	no	library
	Fork/Join framework	mid	task, data	shared	no	library
	TPL	mid	task, data	shared	no	.NET library
	Concurrent Collections	high	data	shared	yes	library
Data Par.	ArBB	mid	data	shared	yes	C++ library
	SAC	high	data	shared	yes	new language
	HPF	high	data	msg pass	no	Fortran extension
	DPH	high	data	shared	yes	Haskell extension/lib
	PLINQ	high	data	shared	yes	.NET library
PGAS	CAF	mid	data, task	PGAS	no	Fortran extension
	UPC	mid	task, data	PGAS	no	C extension
	Fortress	high	data, task	PGAS	no	new language
	Chapel	high	data, task	PGAS	no	new language
	X10	high	data, task	PGAS	no	new language
Par. Decl.	CnC	mid	task, data	shared/msg pass	yes	library
	Parallel Haskells	high	task, data	shared/msg pass	yes/no	extensions/libraries
	Erlang	high	task, data	msg pass (expl.)	no	new language
	Manticore	high	task, data	msg pass	yes/no	SML extension
GPGPU	OpenCL, CUDA	low	data	hierar. mem	no	par comp./seq kernel
	Renderscript	mid	data	hierar. mem	no	C-extension/lib
	C++AMP	mid	data	hierar. mem	no	C++ extension/lib
	Offload	mid	data, task	hierar. mem	no	C++ extension/lib
	SkePU	high	data, task	hierar. mem	yes	library
Skel.	Hadoop MapReduce	high	data	msg pass, shared	yes	library
	P3L	high	task, data	implicit	yes	new language

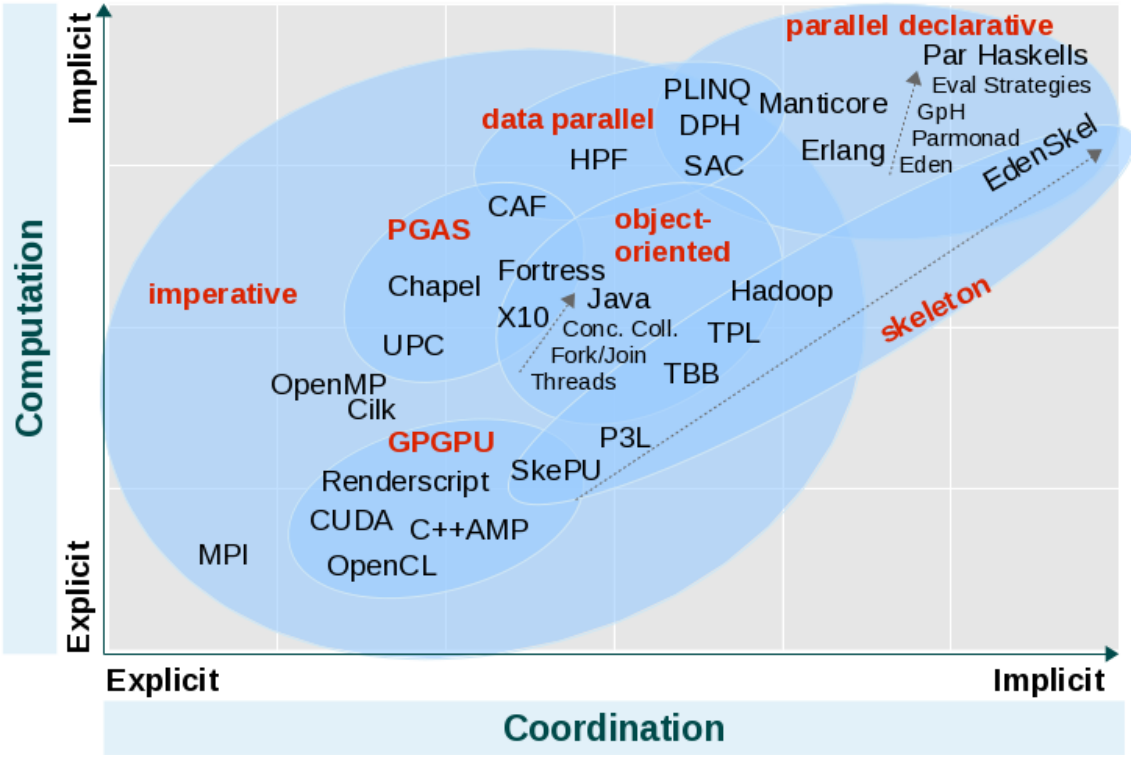

Figure 1: Parallel Programming Models – An Overview

4.2.1. Parallel Imperative

Imperative languages are based on the concepts of state, side-effects, variable manipulation, pointers, iteration, and program counter to control execution and are rather low-level closely matching the uniprocessor architecture [74]. Although low-level of abstraction and explicit control enable manual optimisation and may result in high performance on a single architecture, this approach prevents many automatic optimisations which may result in poor performance across other architectures and reduces portability as well as programmer productivity Nevertheless, these languages are heavily used in the industry [75] and are likely to remain at the core of system-level software, at least in the near future.

4.2.2. Parallel Object-Oriented

Starting from the lowest level, threads are used in object-oriented languages like Java to run jobs in parallel. The programmer is exposed to

the management of threads. Software frameworks such as Fork/Join [76] or TPL [55] abstract over threads and represent independent units of work as tasks with less management involved with manually creating threads. This is left to the RTS which manages a fixed or dynamic pool of threads and automatically maps tasks to running threads. Even more implicit are libraries of concurrent collections which have efficient parallel implementations of operations on common data structures, e.g. arrays and hash tables. These collections hide concurrent access through implicit synchronisation.

4.2.3. Data Parallel

Some languages support only data-parallelism via constructs such as parallel for loop and parallel arrays. This fits a large group of applications where parallelism is identified by domain decomposition. Data parallel languages often provide a sequential model of computation and most coordination aspects are almost completely implicit. However, this model is restrictive and unless the application exhibits data-parallelism, it cannot be used. Most languages in this category efficiently handle regular data-parallelism. DPH [77] is an instance of languages well-suited for irregular data-parallelism using flattening transformation and distributing equal workload to processing units [78]. Data parallel models take advantage of GPUs which are suitable for fine-grained data-parallel computations. For instance, ArBB [79] and SAC [80] can generate vector instructions.

4.2.4. PGAS

The Partitioned Global Address Space (PGAS) abstraction, akin to a tunable virtual-shared memory view, attempts to unify programming by hiding communication and by providing a shared-memory view on potentially physically distributed memory and is becoming increasingly popular in HPC. Extensions to established languages include Unified Parallel C (UPC) [81] and Co-Array Fortran (CAF) [82] and new developments include X10 [83], Chapel [84], and Fortress [85]. Although the level of abstraction is raised, the difficulty of arranging shared-memory accesses re-appears. Moreover, explicitly specifying blocking factors may yield undesirable distributions of shared data that may lead to performance degradation if data locality is impaired, making performance prediction difficult unless the programmer is intimately familiar with the architecture of the underlying target platform.

4.2.5. Parallel Declarative

Declarative languages are based on the concepts of immutability, single assignment, isolated side-effects, higher-order functions, recursion and pattern matching, among others. Due to sophisticated compilation techniques and run-time optimisations available because of their foundation in lambda calculus, declarative programs can deliver competitive performance [86]. Moreover, the performance losses are often offset by productivity gains of the declarative approach that encourages writing portable and high quality code. Most importantly, declarative languages better fit modern parallel architectures, since they allow more flexible coordination of parallel execution and avoid over-specifying evaluation order. For example, Manticore [87] supports multi-level parallelism whereas Haskell offers diverse extensions and libraries [88] to exploit parallelism on multi-cores (GpH), GPUs (Accelerate) and distributed-memory architectures (Eden, Cloud Haskell, HdpH), with Meta-par [89] aiming to unify parallel heterogeneous programming using these models.

4.2.6. GPU Programming

This class of programming models and languages aims at massively data parallel computation. OpenCL, being vendor-independent, targets multi-core CPUs and GPUs, with CUDA targeting only NVIDIA GPUs. At a higher level of abstraction, SkePU provides skeletons building on top of OpenCL and CUDA for GPU and OpenMP for CPU as different backends for heterogeneous computing. Section 5 covers GPU models in more detail.

4.2.7. Algorithmic Skeletons

Algorithmic skeletons [90, 91, 56] encapsulate common parallelism patterns and facilitates structured parallel programming [92, 93] by decoupling the *computational logic* and the *coordination aspects* of a program, and hiding parallelism management from the programmer [94]. This leads to several advantages, such as portability of the parallel behaviour [95] and performance predictability through the application of cost models [96] and sophisticated monitoring techniques [97]. For example, common patterns are *MapReduce*, *TaskFarm*, *Pipeline* and *Divide and Conquer*. These skeletons can be nested, resulting to more advanced skeletal building blocks that implement increasingly complex patterns [98, 99].

5. Heterogeneous Computing

A recent trend in computer architecture design is the integration of specialised hardware such as GPUs, FPGAs and many-core co-processors (such as the Xeon Phi) to create heterogeneous systems, with very different hardware characteristics, such as memory latency, impacting the style of programming on these devices. The combined compute power of these different devices makes it possible to handle the algorithmic complexity and high computational demands of today's increasingly more complex applications. Parallelism, provided through modern heterogeneous systems, and not increased clock rates, has become today's primary source of computational performance [100]. Such systems (e.g. in Figure 2), though, are very complex to efficiently program with, and require the existence of programming models that are powerful enough to allow the users to exploit all the available underlying heterogeneity. This section focuses on such programming models: from lower-level ones such as NVIDIA'S CUDA [101] and OpenCL [102] to higher-level ones such as Microsoft's C++ AMP [103], OmpSs [49], Codeplay's Offload [104], Google's Renderscript [105] and SkePU [106].

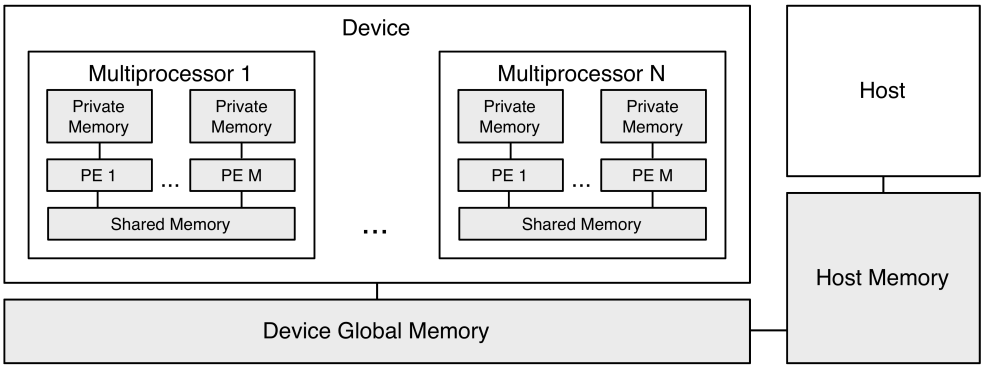

Figure 2: A typical heterogeneous CPU (host)/GPU (device) architecture

5.1. Current, Low-level Heterogeneous Programming Models

Nowadays the most commonly used models for programming heterogeneous devices are NVIDIA's CUDA and OpenCL, an open standard defined by Khronos Group. In our classification, both of these models are low-level,

requiring detailed control of coordination as well as computation, in particular controlling the access to different levels of the memory hierarchy.

The CUDA [101] programming model enables parallel computing on GPU architectures. Developers can write GPU kernel functions in a language such as C, C++ or Fortran in order to issue computational instructions to a GPU. CUDA requires its users to explicitly move data between host and device, and have full access to the device's memory hierarchy (e.g. global, shared and private — see Figure 2). GPUs are different from CPUs in the sense that they emphasise high throughput and consist of thousands of threads, which are able to run in parallel, thus allowing the developers to create powerful solutions that can solve a large variety of data-intensive problems. NVIDIA recently announced that CUDA will be available as a compute model on its upcoming TEGRA 5 architecture for mobile devices, showing that heterogeneous computing is slowly becoming more mainstream.

Open Computing Language (OpenCL) [102] is a heterogeneous programming model targeting existing and emerging architectures that consist of multiple CPUs, GPUs and other accelerators. It is a vendor-independent standard, developed and maintained by the Khronos Group. The OpenCL API is based on a subset of C99 and is very similar to CUDA in the sense that the user writes GPU kernel functions, which are then compiled for the underlying architecture and explicitly offloaded to the requested accelerator device. Through read and write data streams, the available data are transferred to the device, the kernel is executed in parallel and then the results are transferred back to the host. Listing 1 gives an example of a simple OpenCL kernel, performing a vector-vector addition.

Listing 1: A simple OpenCL kernel for vector addition

```
__kernel
void vectorAdd(__global const float* a, __global const float* b,
     __global float* sum, const unsigned int count)
{
  int i = get_global_id(0);

  if (i < count) {
    sum[i] = a[i] + b[i];
  }
}
```

5.2. Emerging, Higher-level Models to Heterogeneous Computing

Current parallel computing models such as CUDA and OpenCL, although powerful tools for exploiting heterogeneity, are too low-level as they provide direct access to the device memory, require a lot of boilerplate initialisation code and are prone to hard to debug errors associated with GPU architectures. Higher-level approaches are necessary towards enabling faster prototyping and ease of programming, maintenance and debugging. The current trend is to move towards heterogeneous programming models that abstract over the underlying GPU hierarchical memory. As an example of this ongoing effort, the next NVIDIA GPU architecture, Maxwell, will introduce unified memory, giving the CUDA programmer direct CPU memory access from the GPU [107].

Microsoft's C++ Accelerated Massive Parallelism (C++ AMP) [103] is an open specification and a DirectX 11 implementation for heterogeneous data parallel computing in C++. The library aims towards performance portability by allowing the same code to run either on a GPU or a CPU. If a GPU is not found in the system then the runtime will send code for execution on the available CPUs. Towards this, Microsoft introduced the *restrict* specifier which can be applied to lambda functions passed inside the C++ AMP system. The compiler then performs a static check on a function that has this specifier to check if the language features inside the lambda can be supported by the available GPU. C++ AMP focuses on data parallelism as the main concept and provides the *parallel_for_each* structure, which defines a parallel loop. The programming model also includes multidimensional arrays, ways to transfer data between the host and the device, and a library of common mathematical functions.

Listing 2: Adding two arrays in C++ AMP

```
void vectorAdd(int n, int * pA, int * pB, int * pSum)
{
    concurrency::array_view<int, 1> a(n, pA), b(n, pB), sum(n,
        pSum);

    concurrency::parallel_for_each(sum.extent, [=](concurrency::
        index<1> i) restrict(amp)
    {
        sum[i] = a[i] + b[i];
    });
}
```

OmpSs [49] is a single-source model that extends OpenMP by introducing new pragmas for programming heterogeneous architectures. The source code is compiled into distinct object files, each one targeting a different processing element of the underlying system, which are scheduled at run time. OmpSs enables incremental parallelisation, where the source code is restructured and optimised step-by-step, while the architecture specific details are separated by the implementation. Unlike OpenCL, OmpSs does not require the programmer to select the concrete back-end. OmpSs is designed to be portable as the same pragmas can be potentially used by any host language and target any architecture that has an implemented backend.

Offload [104] is a single-source programming model by Codeplay for semi-explicitly parallelising C++ code and executing it on a heterogeneous system, such as the Cell processor [108]. The programmer wraps code in an offload block, which is compiled for a backend specific to an accelerating device. The runtime then executes this code asynchronously in a worker thread and sends the results back to the main thread. Any data that is defined inside an offload block resides in device memory, and the compiler is responsible for automatically moving data between the host and the device. This is achieved by having different pointers for host and device at the type level, and by compiling different versions of the same function for the host and the device using a compiler technique known as automatic call-graph duplication. Offload is a higher-level approach that aims to provide performance portability by introducing minimal language extensions, and ease-of-programming by lifting the burden of data management from the programmer.

A trend in mobile devices design is to integrate a multi-core CPU and a GPU on the same chip. Renderscript [105] is a programming model by Google that aims to allow programmers to easily harness the available computing power in such heterogeneous mobile devices. It is part of the Android SDK since the Honeycomb release. Performance portability is the design principle of Renderscript, as it enables the developers to write efficient code that can be executed both on CPU and GPU. The programmer can be agnostic of the underlying architecture as the Renderscript code is compiled during run time on the device. Renderscript abstracts over advanced GPU computing features, such as the underlying GPU memory hierarchy. Although this could potentially hamper application performance, it allows the code to stay hardware-independent, which is important in the mobile landscape with the plethora of different available Android devices.

The highest level of abstraction among the heterogeneous programming

models discussed in this section is provided by SkePU, a C++ template library that abstracts over both parallel computation patterns and hierarchical memory, providing multi-backend skeletons for heterogeneous architectures. The library provides implementations in CUDA and OpenCL for execution on GPUs, and in OpenMP to exploit multicore CPUs. The programmer is able to create his own user-defined functions through a pre-processor macro-based approach. SkePU focuses on skeleton computations over arrays and STL-like vectors, which are a good match for GPU programming. The library provides multiple data-parallel skeletons (Map, Reduce, MapReduce, MapOverlap, MapArray and Scan) and a task-parallel skeleton (Farm). SkePU includes StarPU, a RTS for dynamic scheduling and memory management in heterogeneous architectures. StarPU RTS is capable of scheduling tasks generated by the provided code on a combination of the host and the available accelerating devices.

6. Parallelism Management Policies and Mechanisms

Coordination aspects, as introduced in Section 3, closely correspond to several *policies*, i.e. *"rules that define a choice in the behaviour of a system"* [109], which guide parallel execution. Most relevant parallelism management policies are thread and memory management, communication, load balancing, and monitoring. Generally, policy decisions based on local knowledge lead to more scalable mechanisms, however, often some non-local information is necessary to avoid pathological cases.

This section covers the above policies by introducing several, representative implementation approaches, focusing on those for high-level programming models as discussed in Section 4. While an exhaustive survey of these policies is beyond the scope of this report, we refer to available surveys and taxonomies for further guidance. First, we discuss the negative effects of *overspecification* and review the key coordination *policies*. We then discuss several *mechanisms* to implement these policies at different software layers.

Subsequently, in Table 2 below, we summarise several representative libraries and language run-time systems based on the parallelism identification mechanism, thread management, scheduling, memory management, communication, synchronisation, and load balancing. We then structure our discussion of the concrete mechanisms based on this table.

Overspecification Considered Harmful. Rewriting major sections of applications for each architecture is infeasible and requires a flexible approach to

software adaptation. For instance, fixing task granularity, i.e. overspecifying *evaluation degree*, at implementation or start-up time may lead to load imbalance, if granularity is too coarse, or to excessive overhead, if granularity is too fine [37, 38]. Although chiefly an issue for irregular and dynamic applications, this observation is also relevant if system characteristics change during the execution or if the program is moved to another architecture.

In turn, overspecifying *evaluation order* leads to unnecessary dependencies that reduce potential parallelism and hence diminish scalability. Therefore, architectural trends require embracing the architectural diversity to better exploit parallelism through dynamic policy control. However, to achieve high performance portability the overhead of managing parallelism, change detection, and adaptation should be minimised.

6.1. Thread Management

Thread management involves identification of parallelism and the creation, scheduling, and termination of basic units of execution that include relevant data and associated control structure. For instance, OS-threads comprise state such as register contents and a stack, which makes creation and switching between the threads rather expensive. Usually, a pool of OS-threads is transparently managed and the lightweight threads (or tasks) are multiplexed among the OS-level threads. Several design decisions are involved when managing thread execution across multiple PEs.

A thread can be represented by a data structure on the stack or allocated from the heap. Linked-frame models tend to be more flexible, but incur additional overhead of allocation, de-allocation, and indirection. The implementation of the pool that holds runnable and blocked threads is important too. For instance, lock-free data-structures avoid overhead of explicit locking by exploiting hardware-based atomic operations, e.g. Compare-And-Swap [110].

Parallelism can be identified explicitly by the programmer, or implicitly by using a suitable parallel data structure [80] or high-level skeleton [56], which both restrict the coordination and encapsulate parallelism management. Moreover, explicit identification may be either *mandatory* (e.g. most thread libraries), forcing thread creation (also referred to a eager disconnect), or *advisory* (lazy disconnect), where the annotation is treated as a hint and the final decision rests with the RTS (e.g. GpH). The implicit approach works best for regular problems and those that match the encapsulated patterns, whereas the explicit approach is more general but shifts coordination management either to the programmer, thus lowering productivity, or to the RTS,

Table 2: Overview of Parallelism Management Mechanisms

	Par Ident.	Thread Mgt.	Sched.	Mem Mgt.	Comm.	Sync.	Load Bal.
MPI/PVM	expl.	processes	expl.	expl.	msg pass	messages	expl.
OpenMP	expl. directives	impl.	stat./dyna., mandatory	impl.	SAS[a]	expl. (lock-/fence)	WP[a] (dyna. sched.)
Cilk	expl. primitives	impl.	LIFO	impl.	SAS	expl. (mutex/wait)	WS[a] (FIFO, rand.)
TBB	expl.	impl.	FIFO, unfair	impl.	SAS	expl.	WS (LIFO, rand.)
TPL	expl.	impl.	tunable	impl.	SAS	impl.	WS (tunable)
Java Threads	expl.	expl.	fixed-priority, unfair	impl.	SAS	expl. (lock, mutex, join)	expl.
Fork/Join	expl.	expl.	impl.	impl.	SAS	expl. (join)	WS/expl.
Conc. Collec.	expl.	impl.	impl.	impl.	SAS	impl.	expl.
ArBB	impl.	impl.	stat./dyna., mandatory	impl.	SAS	impl.	WP (dyna. sched.)
SAC	impl.	impl. (arrays)	stat./dyna., advisory	impl.	SAS	impl.	WP (dyna. sched.)
HPF	impl.	impl.	stat.	impl.	SAS	impl.	sched. dependent
DPH	impl.	impl.	impl.	impl.	SAS	impl.	expl.
PLINQ	impl.	impl.	impl.	impl.	SAS	impl.	impl.
CAF	impl.	impl.	impl.	impl. (PGAS)	impl.	expl.	expl. (prescriptive)
UPC/Chapel	impl./expl.[2]	impl.	impl.	impl. (PGAS)	impl.	expl.	expl. (prescriptive)
Fortress/X10	impl./expl.	impl.	impl.	impl. (PGAS)	impl.	expl.	WS
CnC	impl.	impl.	impl.	impl.	SAS	impl.	impl.
GUM(GpH)	annotation	impl. (advi-	FIFO, unfair	impl. (VSM)	impl.	impl.	WS (FIFO, rand.)
GHC-SMP(GpH)	annotation	impl. -sory)	FIFO, unfair	impl.	SAS	impl.	WS (FIFO, rand.)
DREAM(Eden)	expl. process	impl.	roundrobin, fair	impl. (DSM)	impl.	impl.	WP
Manticore	impl./expl.	impl.	FIFO, nestable	impl.	SAS	impl.	WP
OpenCL, CUDA	expl.	impl.	static	expl.	expl.	expl.	expl.
Renderscript	expl.	impl.	impl.	impl.	impl.	expl.	expl.
C++AMP	expl.	impl.	impl.	impl.	impl.	expl.	expl.
Offload	expl.	impl.	impl.	impl.	impl.	expl.	expl.
SkePU	skel.	impl.	tunable	impl.	impl.	impl.	sched. dependent
Hadoop MapRed.	skel.	impl.	tunable	impl.	impl.	impl.	WP (dyna. sched.)
P3L	skel.	impl.	impl.	impl.	impl.	impl.	impl.

[a]SAS: shared address space; WP: work-pushing; WS: work-stealing

[b]UPC, Fortress, Chapel, X10, Manticore: impl. data parallel constructs, expl. task parallelism

complicating its implementation. In contrast to eager disconnect, lazy disconnect allows for dynamic adjustment of the degree of the exploited parallelism by either providing a sequential call with enough information to disconnect and continue execution in a child thread [111] or by inlining child threads in the parent and un-inlining them if necessary [36]. Manual granularity control is not scalable due to rapid architectural evolution and since the behaviour of some applications is dynamic and less predictable. Hence, a mechanism for *thread subsumption* is required to ensure that threads are created only if the benefit of parallel execution outweighs the associated management and communication overhead.

Scheduling [112, 113, 114] determines when and which thread will execute and can be either random, round-robin, queue-based (LIFO, FIFO), depend on a metric e.g. priority-based or heuristic (e.g. Heterogeneous Earliest Finish Time [115]), among others. Moreover, explicit scheduling is the responsibility of the user (e.g. in MPI or pthreads), as opposed to static system scheduling (e.g. OpenMP, UPC), where the schedule is known at compile time, or dynamic system-level scheduling at run time (e.g. Cilk, GpH), which may factor in annotation hints provided by the programmer to judiciously restrict the available parallelism, which is a less demanding task since decomposition is rather implicit and focus is on identification of parallelism. Often the programmer has to select a scheduler from a predefined set and can parameterise it (e.g OpenMP) or even plug-in a custom implementation of a scheduler (e.g. Hadoop), which results in increased flexibility and provides a way to experiment with and tune different scheduling strategies.

6.2. Memory Management

Memory management includes allocation and de-allocation of memory, garbage collection, and maintaining a consistent view of the shared memory. For instance, a virtual shared memory abstraction (as used in GUM [116, 117] a GpH RTS), requires to map local objects that are shared to a global address that allows the remote PEs to locate its contents, whereas distributed-memory view requires using messages (as in DREAM RTS for Eden [118]).

As memory management contributes to the overall management overhead, it is necessary to minimise sharing, to avoid memory fragmentation, to reduce the frequency of disruptive garbage collections (e.g. by replacing stop-the-world mark-and-sweep mechanism by a distributed weight-based or generational garbage collection).

Due to limited scope of this report, we don't include a detailed discussion. Please refer to comprehensive handbooks [119, 120]. The main idea behind efficient parallel garbage collection is to provide mechanisms to reclaim unused memory locally in a decentralised and least disruptive fashion.

6.3. Communication and Synchronisation

Communication is required to share computational load and defines a messaging protocol, packet format, and the serialisation mechanism. It is one of the main sources of overhead, in particular on hierarchical and high-latency architectures. Apart from the network topology, latency and bandwidth are the main parameters that characterise the interconnect and should be taken into account [39, 40].

On distributed-memory architectures messages passing [67, 66] is used, whereas on shared memory architectures data is shared using a shared address space (SAS). PGAS and Virtual Shared Memory (VSM) provide a shared memory view on a potentially distributed set of resources [121, 122]. In either case contention may arise if many messages are send to the same receiver, or many threads attempt to access the same memory location.

Communication overhead can be reduced by hiding latency and sending less but larger messages and by overlapping communication and computation, where possible. Moreover, work sent needs to be substantial to offset communication costs, suggesting that task size information is important. Different mechanisms can be used to protect a shared resource on shared-memory architecture such as semaphores, locks, mutexes, conditional variables, fences, and barriers. Often explicit locking results in contention and creates a bottleneck as the number of cores and threads increases. In some cases, contention on shared memory can be avoided by using wait-free or lock-free data structures, but often it is a matter of scalable design, or even more fundamentally a limitation of the programming model.

6.4. Load Balancing

To exploit parallel architectures, the work needs to be decomposed into sub-tasks and the data to be partitioned to allow simultaneous computation. Subsequently, sub-tasks are placed on PEs and sent off in a way that balances computational load to increase utilisation of the available resources and preserves data locality to reduce communication overhead.

Mapping of sub-tasks to PEs is NP-hard and hence often heuristics or approximations are used. In particular, if an application dynamically generates irregular parallelism or as backload varies on non-dedicated systems, a mechanism is required to dynamically balance the load at run time. Information on the task dependencies and communication patterns can be used to ensure higher data-locality. However, keeping the data local may result in load imbalance, resulting in a trade-off.

Load balancing (or load sharing) aims to distribute work to better utilise the available resources and to satisfy a given goal metric such as minimising run time or power consumption. Migrating executing threads results in a large overhead and should only be considered as a last resort, since predicting effects of migration is nontrivial. Migration may pay off in a situation where a slow machine is engaged in a large computation whilst all other PEs are idle. Additionally, speculative execution may prove useful, as it creates additional opportunities for parallelism at the cost of work duplication.

Load Balancing Taxonomy. Load balancing mechanisms can be subdivided into local and global depending on the effects of the decisions, static or dynamic depending on the decision time (either at start-up or during the execution), distributed, hierarchical, or centralised, based on the availability of holistic information about the system. Moreover, the mechanisms can be either cooperative or non-cooperative depending on whether the underlying scheduling policies are preemptive or not, and optimal or sub-optimal regarding the result. This taxonomy is further refined to classify the sub-optimal approaches as either heuristic or approximative. One further fundamental difference between two large families of load balancing mechanisms is based on whether the producers of work actively distribute it or merely reply to work requests. However, the taxonomy has not been developed with modern parallel architectures in mind and therefore can be extended, depending on the support for heterogeneity and multi-level hierarchy.

Work Pushing. Work pushing, also termed active load distribution, refers to the family of mechanisms, where the generator of work actively pushes work units to other PEs. This approach is common for eager disconnection model and in producer/consumer, pipeline, and task farm patterns (e.g. as used in client/server computing, or in frameworks like MapReduce, or common SPMD-style HPC applications) and suitable if a clear-cut decision can be made at design time that several PEs will rather generate work and others will

take the worker role and receive work units, perform the computation, and return partial results. Several flavours of push-based methods exist, based on the available information (local or global) and on the network topology (e.g. hierarchical).

Work Stealing. On the other extreme of the continuum, work stealing [123, 124, 125] or passive load distribution, refers to a receiver-initiated family of mechanisms, where idle PEs attempt to steal work from other PEs.

Commonly, a steal victim is chosen at *random*, which ensures scalability since no global knowledge is required for the decision. Moreover, a pull-based protocol imposes the overhead on otherwise idle PEs, thus amortising the communication cost. In contrast to a shared-memory architecture, communication costs on a distributed-memory architecture are substantial and work stealing has been demonstrated to benefit from additional information such as system load, latencies, or task size [126, 127, 128].

Apart from the question from which PE to steal, work stealing mechanisms differ in the way a PE responds to a steal request (e.g. by sending the largest work unit, or the one that preserves locality best; or to which PE to forward a request). For instance, on hierarchical systems, it may be preferable to steal work from a local cluster rather than from a remote one, or to send one local and one remote steal request simultaneously. However, predicting the cost-benefit ratio is an open research question and depending on the situation different distribution strategy may perform best.

Cost Modelling. One important technique to predict computation costs of program expressions, and thus to gain important additional information for the scheduler, is to define *cost models* for a particular class of hardware and for a certain programming model. Such cost models provide one solution to the problem of largely varying costs of memory access on heterogeneous hardware, provided that the cost model is sufficiently detailed. For more detailed overview, refer to a survey focussing on parallel hardware [25] and a recent survey paper [34] with a focus on resource analysis.

Influential cost models that have been developed for distributed computation are the Parallel Random Access Machine (PRAM) [129], the Bulk Synchronous Parallel (BSP) [130] and the LogP [131] model. The HLogGP model [132] extends LogP to take heterogeneity into account by using a parameter matrix instead of the scalar parameters that include the speed of compute nodes in a cluster and latency among them.

7. Case Studies

In this section, we present some case studies of parallel applications covering different areas of scientific computing. We start by discussing, in Section 7.1, the parallel implementation of image processing algorithms on a heterogeneous system, using a single-source programming model that also hides details of the memory hierarchy. We refer to implementations of the n-body problem as an important classic HPC application in Section 7.2. Finally, we discuss in Section 7.3 some domain-specific parallel patterns, that exemplify the high-level parallel programming approach, tailored for one particular application domain that has not been covered systematically by HPC so far, namely *symbolic computation*.

While by no means exhaustive, these examples represent challenging applications and the chosen techniques for parallelisation are indicative for current trends in languages and models for parallel computing. Thus, these patterns complement the *"dwarfs/motifs"* identified in [133], representing instances from classic supercomputing parallelism. For a detailed description parallel functional applications the interested reader is referred to [134]. A comparative study of the construction of different large symbolic applications in the parallel functional language GpH is presented in [135].

7.1. Image Processing Filters

Image processing filters are commonly used to enhance the quality of computer graphics. The parallel execution of such image processing algorithms on a heterogeneous system, such as the Cell processor, is presented in [136]. The paper demonstrates the use of Offload C++ technology, a single-source heterogeneous programming model, in five cases of image processing: embossing, sharpening, Laplacian edge detection, grey scaling and noise reduction. Offload technology facilitates parallelisation by hiding the complexity and automating the process of moving data between the host and the device. Thus, the programmer does not have to worry about the underlying memory hierarchy and can write higher-level code which is easier to debug, maintain and extend.

In the case of image sharpening, the algorithm receives a set of pixels, centred around a (x, y) position, as an input, applies the computational kernel and then produces a (modified) pixel, at the same (x, y) position, as an output. Offload semi-explicitly parallelises such computations by offloading the execution of the filter on the available Cell Synergetic Processing Elements

(SPEs). To achieve this, the programmer encapsulates the entire filter code inside an *offload* block, as discussed in Section 5. When the running program reaches an offload block, the runtime spawns a new SPE thread and sends the filter code to be executed on this spawned thread while returning a handler to the main thread. The main thread uses this handler for synchronisation, by waiting for all the running SPE threads to finish the filter computation and then perform a join before continuing with the main thread execution. Experimental results for the image sharpening algorithm showcased speedup of 2.96 over the serial version on 6 SPEs of the Cell processor [136].

7.2. N-Body Simulation

N-Body simulations represent an important class of problems in many areas of science. Imperative languages are efficient at solving the naive algorithm which consists of pairwise comparison between bodies, thus usually requiring two nested loops in the implementation with *in-place* update of the positions and the velocities of the bodies. High-level approaches to implement not only the naive algorithm but also the more advanced Barnes-Hut approximation method have been covered in detail in [88] and [137]. Both papers highlight the ease of expressing several versions of the same or different algorithms in high-level functional languages including Haskell, F# and Scala. Adding parallelism to the problem involved either annotating computations that could usefully be evaluated in parallel or replacing higher-order function e.g. `map` with `parMap` (for which several possible implementations are given in each language) to benefit from parallel execution. The approach demonstrates how functional languages raise the level of abstraction and also enable specifying parallelism in a minimally intrusive way without any major change to the sequential algorithm. The Eden Haskell implementation is also shown to be easily portable on clusters of multicores without any algorithmic changes.

7.3. Symbolic Computation

Typical characteristics of parallel symbolic applications [135, 138] are: they use complex data structures, e.g. trees and graphs rather than flat arrays; they exhibit large degrees of dynamic parallelism, where massive parallelism is generated in bursts throughout the computation; they encompass a high degree of data dependencies, resulting in unpredictable granularities; and they build on symbolic rather than numerical computations, e.g. arbitrary precision integers rather than floats. These characteristics make them

challenging to achieve good speedups, and favour a high-level parallelism approach with an adaptive, dynamic system implementing this model.

7.3.1. An Orbit Pattern

Orbit calculations are common in symbolic computations, in which, a solution space is explored given some initial starting values and a number of generating functions. The Orbit pattern is a frequently occurring task in computational algebra, and has many implementations including the GAP system for computational group theory.

The Orbit pattern is implemented as a function in Haskell, which takes as arguments a list of elements representing the initial state and and a list of generator functions. The pattern will repeatedly apply the generators to the elements of the list and append the results to the list. The process continues until no new elements can be generated for the list without duplication.

The sequential Haskell implementation of the Orbit computation has been parallelised employing a suitable approach for irregular parallelism building on the *workpool* skeleton defined in [139]. Irregularity in the Orbit function is due to the changing set of input as well as the variation of the computational costs for the generator functions based on the input that they are applied to.

Since the sequential code maintains a central queue of inputs that have not yet been processed, a workpool approach for parallelisation effectively distributes this queue to the available processing elements, on a first-come, first-served basis, and then merges the results into the central queue for subsequent processing. The absolute speedups of the orbit calculations, presented in [139], are near-linear (up to a factor of 7.67 on all 8 cores of a multi-core desktop for a set size of 16000).

7.3.2. A Multiple-Homomorphic-Images Pattern

Several computer algebra applications, involving complex data structures, solve problems using an indirect approach. First, the problem is mapped from the input domain to simpler domains, considerably reducing the size of the data structures. Then the problem is solved in these domains, which can be done in parallel, and finally the result in the original domain is reconstructed from these results. The main benefit of this approach for sequential performance comes from the cheaper operations that can be used when solving the problem in these simpler domains, provided the simpler domains preserve basic underlying operations i.e. are homomorphic images.

This approach consists of the following stages: map the input data into several homomorphic images; compute the solution in each of these images, and combine the results of all images to a result in the original domain. All homomorphic images can be computed in parallel in the parallel implementation of this pattern as there is no dependency between them. The combine phase can also be parallelised using a general fold operation. One particular instance of this pattern is a linear system solver discussed in [135].

8. Conclusions

In this survey we have presented a classification of current and emerging programming models for high-level parallel programming, addressing major challenges such as performance portability, scalability to larger clusters and heterogeneity of the underlying hardware. The common theme of these models is to move away from an explicit notion of threads in the program, and to use more structured coordination patterns, instead of the unstructured, explicit synchronisation constructs that have been dominant in low-level models such as MPI. Increasingly, using low-level primitives such as send and receive between explicit threads is considered harmful [65]. Thus, high-level models have to strike a balance between abstraction, to simplify parallel programming, and coordination control, to give a means of parallel performance tuning. Mechanisms to tune the parallelism focus on the distribution of large-scale data structures, in predominantly data-parallel models such as UPC, or annotations in a declarative programming model to tune the granularity of parallelism, such as in parallel Haskell variants.

Developments on the systems implementing these models show some convergence towards work-stealing-based approaches to load balancing and towards highly-tuned, specific patterns of parallel computation on large-scale, distributed architectures. The emergence of increasingly hierarchical architectures, with clusters of clusters of multi-cores, will further add to the heterogeneity of the hardware, and is likely to favour dynamic approaches to work distribution, which avoid being too prescriptive about the possible placement of work in the system and therefore give an adaptive run-time system the required flexibility to adjust its behaviour to the load of the system.

One major source of added heterogeneity are GPUs, which are increasingly used not only in large-scale high-performance computing, but also on a smaller scale in local clusters. Typically, the level of abstraction offered for these processors is even lower than for for standard parallel architectures.

CUDA and OpenCL expose a lot of machine details to the programmer to achieve maximal performance, such as requiring explicit memory management of the GPU hierarchical memory model. We believe that this area will see a similar movement to higher level abstractions in order to make these machines accessible to non-experts in parallel programming. In particular, skeleton-based approaches and specialised data-parallel languages offer the most promising route to increased programmer productivity, while still exploiting the specialised nature of the underlying hardware.

While the proliferation of models and systems for parallel programming might seem intimidating to newcomers to parallel programming, they offer a rich choice of techniques to make best use of the available hardware with variable effort that needs to be spent on parallel programming. In particular, mainstream language extensions, such as OpenMP on physical shared-memory systems and UPC on clusters of multi-cores, provide a good cost-benefit ratio. Skeleton-based approaches, such as Hadoop, offer a high degree of scalability and are currently being extended to a richer class of skeletons, as promoted by the YARN system. Several data parallel languages have achieved very good results on modern GPUs. However, an effective combination of unrestricted task parallelism with such data parallelism still has to be established, and nested data-parallelism remains a challenge.

In general, purely declarative model offers the best fit with parallel computation, due to the absence of any side effects. Today several variants of such languages are available, and we have discussed some of these, in particular parallel Haskell extensions and F#. Other mixed paradigm languages build on the declarative model to introduce parallelism, e.g. Scala, while delineating side-effecting from side-effect-free code in the language. In the long run, such a combination of the conceptually appealing declarative model with the more mainstream object-oriented approach, seems to be the most promising route for high-level parallel programming models.

Acknowledgments

This work is partially funded by SICSA, the Scottish Informatics and Computer Science Alliance. The authors are grateful to the members of the Dependable Systems Group at Heriot-Watt University for useful discussions and support, and to the anonymous reviewers for helpful comments.

References

[1] J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, 5th edition, 2011.

[2] M. J. Flynn, K. W. Rudd, Parallel architectures, ACM Computing Surveys 28 (1996) 67–70.

[3] R. Duncan, A survey of parallel computer architectures, IEEE Computer Survey & Tutorial Series 23 (1990) 5–16.

[4] A. D. Kshemkalyani, M. Singhal, Distributed Computing, Cambridge University Press, 2008.

[5] H. Sutter, The free lunch is over: A fundamental turn toward concurrency in software, Dr. Dobb's Journal 30 (2005).

[6] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, O. O. Storaasli, State-of-the-art in heterogeneous computing, Scientific Programming 18 (2010) 1–33.

[7] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, D. Burger, Dark silicon and the end of multicore scaling, SIGARCH Comput. Archit. News 39 (2011) 365–376.

[8] T. Bjerregaard, S. Mahadevan, A survey of research and practices of network-on-chip, ACM Computing Surveys 38 (2006).

[9] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krueger, A. E. Lefohn, T. J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (2007) 80–113.

[10] I. Kuon, R. Tessier, J. Rose, FPGA Architecture: Survey and Challenges, Found. Trends Electr. Design Automation 2 (2008) 135–253.

[11] D. Ridge, D. Becker, P. Merkey, T. Sterling, Beowulf: Harnessing the power of parallelism in a pile-of-PCs, in: Proceedings of IEEE Aerosspace, 1997, pp. 79–91.

[12] I. Foster, C. Kesselman, The GRID 2: Blueprint for A New Computing Infrastructure, Morgan Kaufmann Publishers, 2nd edition, 2004.

[13] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, A view of cloud computing, Commun. ACM 53 (2010) 50–58.

[14] I. Foster, Y. Zhao, I. Raicu, S. Lu, Cloud computing and grid computing 360-degree compared, in: Grid Computing Environments Workshop, 2008. GCE '08, pp. 1–10.

[15] T. Chen, R. Raghavan, J. N. Dale, E. Iwata, Cell broadband engine architecture and its first implementation – a performance view, IBM Journal of Research and Development 51 (2007) 559–572.

[16] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan, Larrabee: a many-core x86 architecture for visual computing, ACM Trans. Graph. 27 (2008) 18:1–18:15.

[17] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. Yelick, A view of the parallel computing landscape, Commun. ACM 52 (2009) 56–67.

[18] S. Borkar, A. A. Chien, The future of microprocessors, Commun. ACM 54 (2011) 67–77.

[19] M. D. Hill, M. R. Marty, Amdahl's law in the multicore era, Computer 41 (2008) 33–38.

[20] G. Blake, R. Dreslinski, T. Mudge, A survey of multicore processors, Signal Processing Magazine, IEEE 26 (2009) 26–37.

[21] D. Chandra, F. Guo, S. Kim, Y. Solihin, Predicting inter-thread cache contention on a chip multi-processor architecture, Proc. of 11th Intl Symp. on High-Performance Computer Architecture (2005) 340–351.

[22] T. Feng, A survey of interconnection networks, IEEE Computer 14 (1981) 12–27.

[23] S. Pelagatti, Structured Development of Parallel Programs, Taylor & Francis, Inc., Bristol, PA, USA, 1998.

[24] H. Sutter, J. Larus, Software and the concurrency revolution, ACM Queue 3 (2005) 54–62.

[25] D. B. Skillicorn, Foundations of Parallel Programming, volume 6 of *Cambridge International Series on Parallel Computation*, Cambridge University Press, 1994.

[26] I. Foster, Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering, Addison-Wesley, 1995.

[27] C. M. Pancake, Is parallelism for you?, IEEE Computational Science Engineering 3 (1996) 18–37.

[28] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, W. W. Hwu, Program optimization space pruning for a multithreaded GPU, in: Proc. of the 6th Annual Intl Symposium on Code Generation and Optimization, IEEE/ACM, 2008, pp. 195–204.

[29] D. Grewe, M. F. P. O'Boyle, A static task partitioning approach for heterogeneous systems using OpenCL, in: Proc. of the 20th Intl Conf. on Compiler Construction, LNCS 6601, Springer, 2011, pp. 286–305.

[30] A. Collins, C. Fensch, H. Leather, Auto-tuning parallel skeletons, Parallel Processing Letters 22 (2012).

[31] J. K. Ousterhout, Why threads are a bad idea (for most purposes), Presentation at the USENIX'96 Annual Technical Conference, 1996.

[32] J. Manson, W. Pugh, S. V. Adve, The Java memory model, SIGPLAN Not. 40 (2005) 378–391.

[33] E. A. Lee, The problem with threads, IEEE Computer 39 (2006) 33–42.

[34] P. W. Trinder, M. I. Cole, K. Hammond, H.-W. Loidl, G. J. Michaelson, Resource Analyses for Parallel and Distributed Coordination, Concurrency and Computation: Practice and Experience (2013) 309–348.

[35] G. M. Amdahl, Validity of the single-processor approach to achieving large-scale computing capabilities, in: Proceedings of the AFIPS'67 Spring Joint Computer Conference, ACM Press, 1967, pp. 483–485.

[36] E. Mohr, D. Kranz, J. Halstead, R.H., Lazy task creation: a technique for increasing the granularity of parallel programs, Parallel and Distributed Systems, IEEE Transactions on 2 (1991) 264–280.

[37] H.-W. Loidl, Granularity in Large-Scale Parallel Functional Programming, Ph.D. thesis, Department. of Computing Science, University of Glasgow, 1998.

[38] K. Hammond, Why parallel functional programming matters: Panel statement, in: A. Romanovsky, T. Vardanega (Eds.), Ada-Europe 2011, volume 6652 of *LNCS*, pp. 201–205.

[39] Z. Li, P. H. Mills, J. H. Reif, Models and resource metrics for parallel and distributed computation, in: Proc. of the 28th Hawaii Intl Conference on System Sciences (HICS'95), IEEE, 1995, pp. 133–143.

[40] B. Maggs, L. Matheson, R. Tarjan, Models of parallel computation: A survey and synthesis, in: Proc. of the 28th Hawaii Intl Conference on System Sciences (HICSS'95), IEEE, 1995, pp. 61–70.

[41] R. D. Blumofe, C. E. Leiserson, Scheduling multithreaded computations by work stealing, J. ACM 46 (1999) 720–748.

[42] J. Kephart, D. Chess, The vision of autonomic computing, Computer 36 (2003) 41–50.

[43] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, F. Zambonelli, A survey of autonomic communications, ACM Trans. Auton. Adapt. Syst. 1 (2006) 223–259.

[44] K. Astrom, Adaptive feedback control, Proceedings of the IEEE 75 (1987) 185–217.

[45] C.-T. Yang, C.-L. Huang, C.-F. Lin, Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters, Computer Physics Communications 182 (2011) 266–269.

[46] R. Rabenseifner, Hybrid Parallel Programming on HPC Platforms, Proc. European Workshop on OpenMP '03 (2003).

[47] R. Rabenseifner, G. Hager, G. Jost, Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes, in: Proc. of the 17th Euromicro Intl Conf. on Parallel, Distributed and Network-based Processing, PDP '09, IEEE Computer Society, 2009, pp. 427–436.

[48] A. Varbanescu, P. Hijma, R. V. van Nieuwpoort, H. Bal, Towards an effective unified programming model for many-cores, in: Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW), 2011 IEEE International Symposium on, pp. 681–692.

[49] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, OmpSs: a proposal for programming heterogeneous multi-core architectures, Parallel Processing Letters 21 (2011) 173–193.

[50] J. Berthold, H.-W. Loidl, A. A. Zain, Scheduling light-weight parallelism in ArTCoP, in: D. S. Warren, P. Hudak (Eds.), Proceedings of the Conference on Practical Aspects of Declarative Languages (PADL'08), volume 4902 of *LNCS*, Springer, 2008, pp. 214–229.

[51] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, A. Singhania, The Multikernel: A new OS architecture for scalable multicore systems, in: ACM Symposium on OS Principles, ACM, 2009.

[52] N. Shavit, Data structures in the multicore age, Commun. ACM 54 (2011) 76–84.

[53] G. L. Steele, Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful, SIGPLAN Not. 44 (2009) 1–2.

[54] J. Diaz, C. Munoz-Caro, A. Nino, A survey of parallel programming models and tools in the multi and many-core era, Parallel and Distributed Systems, IEEE Transactions on 23 (2012) 1369–1386.

[55] D. Leijen, W. Schulte, S. Burckhardt, The Design of a Task Parallel Library, in: OOPSLA'09 – Intl Conf. on Object Oriented Programming Systems Languages and Applications, ACM Press, 2009, pp. 227–242.

[56] H. González-Vélez, M. Leyton, A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers, Software: Practice and Experience 40 (2010) 1135–1160.

[57] D. R. Butenhof, Programming with POSIX Threads, Addison-Wesley, 2nd edition, 1997.

[58] S. Oaks, H. Wong, Java Threads, O'Reilly, 2nd edition, 1999.

[59] B. Chapman, G. Jost, R. van der Pas (Eds.), Using OpenMP: Portable Shared Memory Parallel Programming, MIT Press, 2007.

[60] M. Herlihy, J. E. B. Moss, Transactional memory: architectural support for lock-free data structures, SIGARCH Comput. Archit. News 21 (1993) 289–300.

[61] T. Harris, A. Cristal, O. S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, M. Valero, Transactional memory: An overview, IEEE Micro 27 (2007) 8–29.

[62] T. Harris, J. Larus, R. Rajwar, Transactional Memory, Morgan and Claypool Publishers, 2nd edition, 2010.

[63] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, S. Chatterjee, Software transactional memory: Why is it only a research toy?, ACM Queue 6 (2008) 46–58.

[64] A. Dragojević, P. Felber, V. Gramoli, R. Guerraoui, Why STM can be more than a research toy, Commun. ACM 54 (2011) 70–77.

[65] S. Gorlatch, Send-receive considered harmful: Myths and realities of message passing, ACM Transactions on Programming Languages and Systems (TOPLAS) 26 (2004) 47–56.

[66] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM: Parallel Virtual Machine: A User's Guide and Tutorial for networked Parallel Computing, MIT Press, 1994.

[67] M. Snir, S. W. Otto, D. W. Walker, J. J. Dongarra, S. Huss-Lederman, MPI: The Complete Reference, MIT Press, 1995.

[68] C. Hewitt, P. Bishop, R. Steiger, A universal modular ACTOR formalism for artificial intelligence, in: Proceedings of the 3rd Intl Joint Conf. on Artificial Intelligence, Morgan Kaufmann, 1973, pp. 235–245.

[69] G. Agha, ACTORS: A Model of Concurrent Computation in Distributed Systems, MIT Press, 1986.

[70] S. Marlow, R. Newton, S. Peyton Jones, A monad for deterministic parallelism, in: Proceedings of the 4th ACM symposium on Haskell, Haskell '11, ACM, New York, NY, USA, 2011, pp. 71–82.

[71] P. Maier, P. Trinder, Implementing a high-level distributed-memory parallel Haskell in Haskell, in: Proc. of the 23rd Intl Conf. on Impl. and Application of Func. Lang., IFL'11, Springer, 2012, pp. 35–50.

[72] D. Gelernter, N. Carriero, Coordination languages and their significance, Commun. ACM 35 (1992) 96–107.

[73] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, P. Trinder, Seq no more: better strategies for parallel Haskell, in: Proceedings of the 3rd Symposium on Haskell, Haskell '10, ACM, 2010, pp. 91–102.

[74] P. Van Roy, Programming paradigms for dummies: What every programmer should know, New Computational Paradigms for Computer Music (2009) 9–47.

[75] TIOBE Software, TIOBE programming community index, www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2013.

[76] D. Lea, A Java fork/join Framework, in: Java'00 — ACM 2000 Conference on Java Grande, ACM Press, 2000, pp. 36–43.

[77] S. L. Peyton Jones, R. Leshchinskiy, G. Keller, M. M. T. Chakravarty, Harnessing the multicores: Nested data parallelism in Haskell, FSTTCS, 2008, pp. 383–414.

[78] G. E. Blelloch, NESL: A Nested Data-parallel Language (version 3.1), Technical Report CMU-CS-95-170, Carnegie Mellon University, 1995.

[79] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, et al., Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language, in: 9th Annual International Symposium on Code Generation and Optimization, IEEE/ACM, 2011, pp. 224–235.

[80] C. Grelck, S.-B. Scholz, SAC – a functional array language for efficient multi-threaded execution, International Journal of Parallel Programming 34 (2006) 383–427.

[81] UPC Consortium, Unified Parallel C Langauge Spec. v1.2 LBNL-59208, Technical Report, Lawrence Berkeley National Lab, 2005.

[82] R. W. Numrich, J. Reid, Co-Array Fortran for parallel programming, ACM SIGPLAN Fortran Forum 17 (1998) 1–31.

[83] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, X10: An object-oriented approach to non-uniform cluster computing, in: Proceedings of OOPSLA'05, ACM Press, 2005, pp. 519–538.

[84] B. L. Chamberlain, D. Callahan, H. P. Zima, Parallel programmability and the Chapel language, International Journal of High Performance Computing Applications 21 (2007) 291–312.

[85] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., S. Tobin-Hochstadt, The Fortress Language Specification Version 1.0, Technical Report, Sun Microsystems, Inc., 2008.

[86] G. Mainland, S. Peyton Jones, S. Marlow, R. Leshchinskiy, Haskell beats C using generalised stream fusion, in: ICFP'13, Submitted, 2013.

[87] M. Fluet, M. Rainey, J. Reppy, A. Shaw, Y. Xiao, Manticore: A heterogeneous parallel language, in: DAMP 2007: Workshop on Declarative Aspects of Multicore Programming, ACM Press, 2007, pp. 37–44.

[88] P. Totoo, H.-W. Loidl, Parallel Haskell implementations of the N-body problem, Concur. and Comp.: Practice and Experience (to appear).

[89] A. Foltzer, A. Kulkarni, R. Swords, S. Sasidharan, E. Jiang, R. Newton, A meta-scheduler for the Par-Monad: composable scheduling for the heterogeneous cloud, in: Proc. of the 17th ACM SIGPLAN Intl Conf. on Functional programming, ICFP '12, ACM, 2012, pp. 235–246.

[90] M. Cole, Algorithmic skeletons: structured management of parallel computation, Pitman, 1989.

[91] F. Rabhi, S. Gorlatch, Patterns and skeletons for parallel and distributed computing, Springer-Verlag New York Inc, 2003.

[92] M. Ghanem, Structured Parallel Programming Using Performance Models and Skeletons, Ph.D. thesis, Department of Computing, Imperial College, 1999.

[93] J. Yang, Co-ordination Based Structured Parallel Programming, Ph.D. thesis, Department of Computing, Imperial College, 1998.

[94] D. B. Skillicorn, Towards a higher level of abstraction in parallel programming, In: Proc. of Programming Models for Massively Parallel Computers (1995) 78–85.

[95] J. Darlington, M. Ghanem, H. To, Structured parallel programming, In: Proc. Programming Models for Massively Parallel Computers (1993) 160–169.

[96] M. Hamdan, A survey of cost models for algorithmic skeletons, Technical Report RM/11/99, Heriot-Watt University, 1999.

[97] H. González-Vélez, M. Cole, Adaptive structured parallelism for distributed heterogeneous architectures: A methodological approach with pipelines and farms, Concur.: Pract. and Exper. 22 (2010) 2073–2094.

[98] J. Darlington, Y. Guo, H. To, J. Yang, Parallel skeletons for structured composition, ACM SIGPLAN Notices 30 (1995) 19–28.

[99] G. Michaelson, N. Scaife, P. Bristow, P. King, Nested algorithmic skeletons from higher order functions, Parallel Algorithms and Applications 16 (2001) 181–206.

[100] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, V. Volkov, Parallel computing experiences with CUDA, Micro, IEEE 28 (2008) 13–27.

[101] NVIDIA Corporation, CUDA programming guide 2.0, 2008.

[102] Khronos OpenCL Working Group, The OpenCL specification, version 1.1, revision 44, 2011.

[103] K. Gregory, A. Miller, C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++, Microsoft Press, 2012.

[104] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, G. Russell, Offload–automating code migration to heterogeneous multicore systems, in: High Performance Embedded Architectures and Compilers, Springer, 2010, pp. 337–352.

[105] Google Inc., Renderscript, http://developer.android. com/guide/topics/renderscript/compute.html, 2011.

[106] J. Enmyren, C. Kessler, SkePU: a multi-backend skeleton programming library for multi-GPU systems, in: Proc. of the 4th Intl Workshop on High-Level Parallel Programming and Applications, ACM, pp. 5–14.

[107] J.-H. Huang, NVIDIA Keynote, GPU Technology Conference, 2013.

[108] H. P. Hofstee, Power efficient processor architecture and the Cell processor, in: High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on, IEEE, pp. 258–262.

[109] M. Sloman, Policy driven management for distributed systems, Journal of Network and Systems Management 2 (1994).

[110] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Elsevier, 2012. Revised first edition.

[111] S. C. Goldstein, K. E. Schauser, D. E. Culler, Lazy threads: implementing a fast parallel call, J. Par. Distrib. Comput. 37 (1996) 5–20.

[112] T. Casavant, J. Kuhl, A taxonomy of scheduling in general-purpose distributed computing systems, Software Engineering, IEEE Transactions on 14 (1988) 141–154.

[113] F. Dong, S. G. Akl, Scheduling Algorithms for Grid Computing: State of the Art and Open Problems, TR 2006-504, School of Computing, Queens University, Kingston, Ontario, 2006.

[114] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, M. Prieto, Survey of scheduling techniques for addressing shared resources in multicore processors, ACM Comput. Surv. 45 (2012) 4:1–4:28.

[115] H. Topcuoglu, S. Hariri, M.-y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, Parallel and Distributed Systems, IEEE Transactions on 13 (2002) 260–274.

[116] P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, S. L. Peyton Jones, GUM: a portable parallel implementation of Haskell, in: Proceedings of the PLDI'96, ACM Press, 1996, pp. 79–88.

[117] H.-W. Loidl, The virtual shared memory performance of a parallel graph reducer, in: Proc. of the Intl Symposium on Cluster Computing and the Grid (CCGrid/DSM 2002), IEEE, 2002, pp. 311–318.

[118] R. Loogen, Y. Ortega-Mallén, R. Peña-Marí, Parallel functional programming in Eden, J. Funct. Program. 15 (2005) 431–475.

[119] R. Jones, R. Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Research Monographs in Parallel and Distributed Computing, John Wiley and Sons, Inc, New York, 1996.

[120] R. Jones, A. Hosking, E. Moss, The Garbage Collection Handbook: The Art of Automatic Memory Management, Chapman & Hall/CRC, 1st edition, 2011.

[121] L. Iftode, J. P. Singh, Shared virtual memory: Progress and challenges, Proceedings of the IEEE 87 (1999) 498–507.

[122] B. Nitzberg, V. Lo, Distributed shared memory: A survey of issues and algorithms, Computer 24 (1991) 52–60.

[123] F. Burton, M. Sleep, Executing functional programs on a virtual tree of processors, in: Proceedings of the Conference on Functional Program Language and Computer Architecture, ACM, 1981, pp. 187–194.

[124] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou, Cilk: An efficient multithreaded runtime system, in: Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP'95), pp. 207–216.

[125] M. Frigo, C. E. Leiserson, K. H. Randall, The implementation of the Cilk-5 multithreaded language, SIGPLAN Not. 33 (1998) 212–223.

[126] R. V. van Nieuwpoort, T. Kielmann, H. E. Bal, Efficient load balancing for wide-area divide-and-conquer applications, ACM SIGPLAN Not. 36 (2001) 34–43.

[127] A. Al Zain, Implementing High-Level Parallelism on Computational GRIDs, Ph.D. thesis, Heriot-Watt University, 2006.

[128] V. Janjic, Load Balancing of Irregular Parallel Applications on Heterogeneous Computing Environments, Ph.D. thesis, School of Computer Science, University of St Andrews, 2011.

[129] S. Fortune, J. Wyllie, Parallelism in random access machines, in: Proc. of the Symposium on Theory of Computing, ACM, 1978, pp. 114–118.

[130] L. G. Valiant, A bridging model for parallel computation, Communications of ACM 33 (1990) 103–111.

[131] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, T. von Eicken, LogP: Towards a realistic model of parallel computation, Proceedings of the 4th ACM Symposium on Principles and Practice of Parallel Programming (1993) 1–12.

[132] J. Bosque, L. Pastor, A parallel computational model for heterogeneous clusters, IEEE Trans. on Parallel and Distributed Systems 17 (2006).

[133] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, The Landscape of Parallel Computing Research: A View from Berkeley, Technical Report UCB/EECS-2006-183, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[134] K. Hammond, G. Michelson (Eds.), Research Directions in Parallel Functional Programming, Springer-Verlag, London, UK, UK, 2000.

[135] H.-W. Loidl, P. W. Trinder, K. Hammond, S. B. Junaidu, R. G. Morgan, S. L. P. Jones, Engineering Parallel Symbolic Programs in GpH, Concurrency: Practice and Experience 11 (1999) 701–752.

[136] A. F. Donaldson, U. Dolinsky, A. Richards, G. Russell, Automatic offloading of C++ for the Cell BE processor: a case study using Offload,

in: Intl Conf. on Complex, Intelligent and Software Intensive Systems, IEEE, 2010, pp. 901–906.

[137] P. Totoo, P. Deligiannis, H.-W. Loidl, Haskell vs. F# vs. Scala: A High-level Language Features and Parallelism Support Comparison, in: Proceedings of the Workshop on Functional High-Performance Computing, ACM, 2012, pp. 49–60.

[138] R. H. Halstead, Multilisp: a language for concurrent symbolic computation, ACM Transactions on Programming Languages and Systems 7 (1985) 501–538.

[139] U. Klusik, R. Loogen, S. Priebe, F. Rubio, Implementation skeletons in Eden: Low-effort parallel programming, in: IFL '00: Selected Papers from the 12th International Workshop on Implementation of Functional Languages, Springer-Verlag, London, UK, 2001, pp. 71–88.