Static Macro Data Flow: Compiling Global Control into Local Control

Pritish Jetley and Laxmikant V. Kalé,
Department of Computer Science,
University of Illinois at Urbana-Champaign, USA
{pjetley2, kale}@illinois.edu

***Abstract*—The expression of parallel codes through abstract, high-level specifications of global control and data flow can greatly simplify the task of creating large parallel programs. We discuss the challenges of compiling such global flows into the behavioral descriptions of individual component objects in an SPMD environment. We present our work in the context of Charisma, a language that describes global data and control flow through a simple script-like language. Inter-object interactions are realized through the *production* and *consumption* of data. The compiler infers communication patterns between objects and generates appropriate messaging code. We discuss the productivity and performance benefits of compiling such global specifications into local descriptions of control flow embodied by a language called Structured Dagger (SDAG).**

I. Introduction

Traditional parallel programming models fail to strike a favorable balance between productivity and performance. The issue of productivity, though often overlooked, is important nonetheless. With the advent of multicore desktops and the dawning of a new era of scientific investigation, one centered around computer simulations developed to refine our understanding of physical phenomena, the base of developers expected to program in parallel has grown rapidly. But engineering parallel applications is hard, and given their varied backgrounds and degrees of familiarity with parallel programming, users of parallel systems must be provided with tools, models and environments that ease the burden of programming in parallel.

Charisma [1] is a parallel language designed to address these concerns. It aims to capture the expression of a specific class of parallel programs, namely those typified by static data flow (SDF). By restricting the scope of expression of the language, Charisma provides elegant syntax and simple semantics in which static data flow programs may be written. Charisma programs describe the global control and data flow between members of parallel object arrays through a high-level *orchestration* notation. Objects communicate with static sets of "neighbors" through the *production* and *consumption* of data items. The compiler parses orchestration statements to infer (static) communication patterns between objects. It then generates CHARM++ code to realize these patterns.

In this paper, we discuss the process by which the global Charisma specifications of data and control flow are translated into a language called Structured Dagger (SDAG). Structured Dagger is a *local coordination* language for the ordering of events such as asynchronous message receipts on parallel objects comprising a message-driven system such as CHARM++ [2]. In other words, we present a way to translate the global control and data flow of parallel SDF programs into their local counterparts. While programming in abstract representations such as Charisma can increase productivity in itself, this paper demonstrates that our techniques of converting global flows to local ones afford additional productivity and performance benefits. Although we discuss the theme in terms of specific languages, the principles of translation listed herein are applicable to other SDF languages and their conversion to reactive, message-driven specifications of object behavior.

We begin by introducing Charisma and SDAG. We then describe the merits and issues of translating global flows to local ones. A comparison of the current Charisma compiler infrastructure, that emits low-level CHARM++ code, and our efforts at translation of Charisma into SDAG follows. We end with a study of the performance of code generated by the two strategies. Our experiments involve algorithms of some significance in numerical simulations and scientific computing. We study the performance of Cannon's algorithm for matrix multiplication, a transpose-based three dimensional FFT and a five-point stencil computation.

II. Global Flows and Charisma

Charisma embodies the concepts of static dataflow in a high-level language. In the vein of CHARM++, it specifies computation in terms of collections of *objects*, called *object arrays*. It provides a high-level notation to coordinate the behavior of these collections, which is why it is referred to as an *orchestration* language. The use of objects as the basic computational units engenders coarse-grained dataflow.

Further, Charisma enforces a clear separation between the serial and parallel parts of a computation. The orchestration code details the interactions between objects in the form of statements that *produce* and *consume* data. Crucially, it provides a global view of the flow of control and data. The compiler parses this flow information to determine patterns of communication between objects. Objects have separate serial specifications comprising C++ methods. Through the use of special keywords such as `produce` within these methods, objects are able to publish the results of serial computation without regard to their destination. The compiler incorporates the sequential code into the communication/synchronization skeleton generated via orchestration statements to arrive at a CHARM++ program.

Language Features. We familiarize the reader with programming in Charisma by providing a brief discussion of the syntax and semantics of the language. Fragments of example code accompany these descriptions; a more detailed text on the language would be [1].

A. *The* `foreach` *Construct*

Users specify the invocation of a single entry method across all objects through the `foreach` construct. These invocations are independent of each other and depend on the availability of consumed parameters at the various objects in question. Enclosed statements produce and consume suitably indexed *parameters*. In the boxed example code, each of a collection of objects, `w[i]`, produces a parameter `p[i]` that shares its index, and consumes a parameter `p[i-1]` produced by the "neighboring" object `w[i-1]`.

```
foreach i in w
 (p[i]) <- w[i].prod(...);
 (...) <- w[i].cons(p[i-1]);
end-foreach
```

The compiler generates code to send messages from object `w[i]` to `w[i+1]` for each `i`.

B. *Control Flow*

Loops are specified through `for` and `while` statements. The code below shows the `f` method of each object being called repeatedly. During the first iteration, objects consume parameter values generated by statements before the loop in the program text.

```
for iter = 1 to MAX_ITER
 foreach i in w
  (p[i]) <- w[i].f(p[i-1]);
 end-foreach
end-for
```

Every successive iteration consumes parameters produced by the previous one. Values produced during the final iteration of the loop act as inputs to statements following the loop.

C. *Execution Semantics*

A deterministic execution guarantee eases the burden of developers by providing a simple semantic infrastructure within which to develop and reason about programs. It ensures reproducibility of results and faults in programming logic, thereby making programs easier to debug and maintain.

Although the underlying CHARM++ infrastructure provides no message ordering guarantees, Charisma uses the notions of *data dependence* and *program order* to form a deterministic execution model. The first of these requires that the method associated with a statement be invoked on an object only after it has received all the parameters it consumes. The latter states that the order in which methods are invoked on individual objects is determined by their positions in the program text relative to each other.

Together, these two clauses give Charisma deterministic semantics. Frequently, however, there arise cases when the lack of any dependences between actions allows their concurrent execution. In such cases, the program order clause would preclude any desirable overlap of computation and communication between them. Statements that have all of their consumed parameters available may be stalled by preceding ones in program order, even though there is no real dependence between them. In Charisma, such concurrency is expressed through the `overlap` statement.

```
for iter = 1 to MAX_ITER
 foreach x,y in cells
  (NW[x,y],...,SE[x,y])
     <- cells[x,y].prodParts();
 end-foreach
 overlap
  foreach x,y in cells
   cells[x,y].selfForces();
   cells[x,y].forces(NW[x+1,y-1]);
   ...
   cells[x,y].forces(SE[x-1,y+1]);
  end-foreach
 end-overlap
end-for
```

Consider a molecular dynamics (MD) code that performs spatial decomposition among processors, and allows the accretion of forces on a cell from particles in its surrounding neighbors in any order.

No ordering constraints are placed on overlapped statements; they may execute as the parameters they consume become available. The boxed code above is an MD simulation where the various components of the non-bonded force on a cell `cells[x,y]` are computed concurrently.

III. Structured Dagger

Structured Dagger (SDAG) is a coordination language that allows the clear expression of control flow within message-driven CHARM++ objects. Like CHARM++, SDAG programs specify interfaces of objects that define their behaviors. These interfaces consist of non-preemptible *entry* methods, which specify pieces of computation (`atomic` blocks), intra-object data dependences (`when` statements), control flow (`for`, `while`, `if` constructs, etc.) and the sequence in which such statements are ordered. A more detailed description of the language may be found in [3]. We explain some of these constructs in terms of a two-dimensional Jacobi relaxation computation with blocked decomposition written in SDAG.

```
entry void jacobi(){
  for(I = 1; I <= MAX_ITER; I++)
    atomic{ sendBoundaries(); }
    when recv_lb[I](msg *l),
         recv_rb[I](msg *r),
         recv_ub[I](msg *u),
         recv_bb[I](msg *b){
      atomic{ stencil(l,r,u,b); }
    }
}
```

The boxed code above specifies an entry method `jacobi` which defines all control and data flow local to an object. The central construct of SDAG is the `when` statement. This keyword specifies local data dependences, associating specific actions with the occurrence of events such as data receipt. Therefore, `when` statements are a way of specifying the reactive behavior of an object. These statements are executed in a two-phase manner. The first step involves the scheduling of high-level instructions so as to preserve program order. During this phase, *triggers* are set up to acknowledge the readiness of an instruction to consume data. If these data are available, the actions associated with their receipt are carried out. Otherwise, they take place at a later time via a callback mechanism. The actions themselves are represented by `atomic` constructs, which are akin to basic blocks of compiler nomenclature and form a unit of computational work. An `atomic` block may not enclose any SDAG constructs.

In the Jacobi program at hand, the `for` construct sets up `MAX_ITER` iterations of the computation. The `sendBoundaries` and `stencil` methods enclosed within `atomic` blocks represent the generation of boundaries and the Jacobi relaxation operation performed on local data respectively. The method `sendBoundaries` includes code to send its various boundaries to the corresponding neighbors by invoking the `recv_rb` method on its left neighbor, `recv_lb` on its right neighbor, etc. The `stencil` computation may commence only upon the receipt of boundary data from all four neighbors. These dependences are enforced by the `when` construct. The `when` statement also includes annotations within square brackets (`[I]`) involving the loop index variable `I`. Such annotating expressions are evaluated at run time and checked against tags included within received messages to ensure that they are delivered to the correct iteration. In the context of the present SDAG program, this means that data are sent to the correct instances of `stencil`.

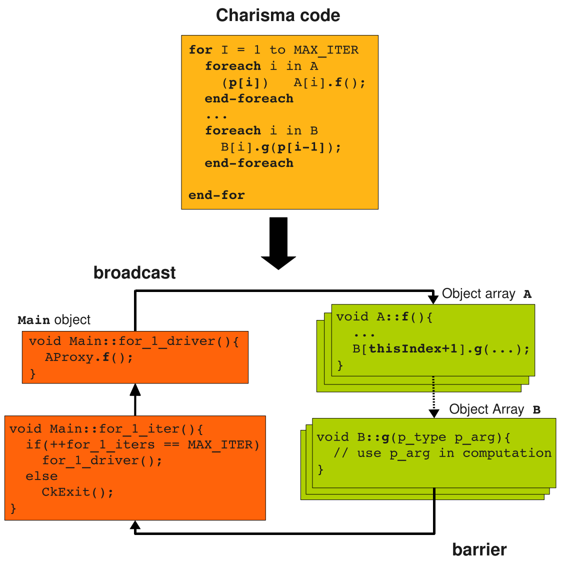
IV. Compiling Global Flows

Converting a Charisma program into its corresponding CHARM++ code entails the translation of a global view of control, one which involves inter-object communication and synchronization, into reactive specifications that are local to the parallel objects they describe. Certain key issues arise with regard to this process of translation. We touch upon each of these issues in turn. The discussion is conducted in the context of a fragment of example orchestration code, depicted in Figure 1. The figure illustrates a `for` loop that encloses `foreach` statements invoked on different object arrays. A data dependence exists between methods `f` and `g` of arrays `A` and `B` respectively, since the invocations of the former produce elements of parameter `p` and those of the latter consume them. Throughout the discussion, we make references to the techniques used by the previous compiler to translate the global specifications of Charisma into CHARM++.

A. Projecting global control onto objects

Charisma orchestration code provides a single body of source that interleaves the high-level instruction sequence of each of the constituent objects of the program. However, CHARM++ programs are written in terms of reactive specifications of *individual* objects, so that in translating from the former to the latter, we must project global flows onto code meant for different object types.

When translating to CHARM++, this requires that we extract methods from `foreach` statements and include them as entry methods of the corresponding objects. With reference to Figure 1, the two `foreach` statements are translated into methods `A::f` and `B::g`.



```
for I = 1 to MAX_ITER
  foreach i in A
    (p[i])   A[i].f();
  end-foreach
  ...
  foreach i in B
    B[i].g(p[i-1]);
  end-foreach

end-for
```

```
void Main::for_1_driver(){
  AProxy.f();
}
```

```
void Main::for_1_iter(){
  if(++for_1_iters == MAX_ITER)
    for_1_driver();
  else
    CkExit();
}
```

```
void A::f(){
  ...
  B[thisIndex+1].g(...);
}
```

```
void B::g(p_type p_arg){
  // use p_arg in computation
}
```

Fig. 1. The translation of a global flow specification into CHARM++ code. Iterations of the `for` loop are driven by successful evaluation of the associated predicate on the `Main` object. Objects continue only upon receipt of notification from it.

B. Enforcing data dependences

Charisma defines communication between objects in terms of patterns of production and consumption of data. Correct translation of the global flow of data, therefore, hinges in large part upon the interpretation of these patterns by the compiler. The Charisma compiler looks for forward and loop-carried dependences by matching names of parameters that are produced and consumed at various points in the program. Objects may only produce elements of the parameter space that share their identifying index. The communication pattern is implied by the indices of consumed parameters. To translate this symmteric style of communication into the asymmetric, sender-centric mode of CHARM++, the compiler inverts the (restricted) affine function that forms the index of the consumed parameter. This inverted index is used by the sender to address the particular object receiving the element. In the example code of Figure 1, the dotted arrows show the flow of data between arrays `A` and `B`. Each `A[i]` produces an element `p[i]`, whereas each `B[i]` consumes `p[i-1]`. The compiler interprets this as the invocation of `g` by `A[i]` on `B[i+1]`.

C. Program order and control flow

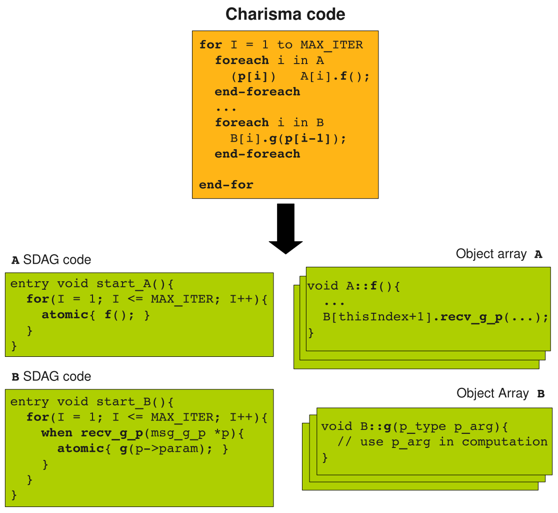
Finally, the compiler must attend to the conversion of global flow of control into parallel object code that effects the intended semantics of constructs such as the `for`, `while` and `if` statements. The Charisma compiler uses a *centralized* evaluation of the predicates associated with the statements of these constructs. Consider the execution of a Charisma `for` loop. Control starts with the main object, which sends notifications to the relevant arrays, commencing the computation. Any data dependences that must be satisfied in the code that follows are fulfilled through the code generation process described previously. The execution of the loop sink statement is followed by a barrier which transfers control back to the main object. The predicate guarding the loop body is evaluated on the main object once again, and control is conditionally transferred to either the first statement in the body or the first statement following the `for`. Statements involving the `while` and `if` constructs are handled similarly.

The various stages of control flow in a Charisma program are highlighted in Figure 1. Notice in particular the role of the `Main` object: every iteration begins with a broadcast from `for_1_driver` to elements of `A` through the asynchronous invocation of entry method `f`. Control returns to the main object by the barrier following the consumption of `p` by members of `B`.

V. FROM CHARISMA TO SDAG

There are certain advantages to compiling a higher-level language such as Charisma to one like SDAG, since the latter allows the **succinct expression of local control** and also has substantial run-time and compiler support from the CHARM++ system. Moreover, there is **less impedance mismatch** between Charisma and SDAG than between Charisma and CHARM++, since both Charisma and SDAG deal with abstract and explicit representations of dependence and flow, whereas these are implicit in the asynchronous method invocations of CHARM++ programs. Further, a layered approach to the translation of an abstract language benefits from **existing infrastructure** for the conversion of the intermediate representation itself. For example, when translating directly to CHARM++, the Charisma compiler must be careful to emit message sequencing code that prevents messages from crossing iterations of `for` loops. We simply leverage the annotated `when` constructs of SDAG for this purpose. Writing compilers for restricted, special purpose languages such as Charisma therefore becomes an easier task. Finally, optimizations can be effected at an intermediate level of representation: high-level languages tend to be abstract and hide opportunities for implementation-level optimizations, whereas low-level specifications obscure important semantic information such as data and control flow. In such a situation, an intermediate target becomes a conduit information flow from higher to lower levels of abstraction.

We now describe a methodology for the translation of global flow as represented by a language such as



```
for I = 1 to MAX_ITER
  foreach i in A
    (p[i])   A[i].f();
  end-foreach
  ...
  foreach i in B
    B[i].g(p[i-1]);
  end-foreach

end-for
```

```
entry void start_A(){
  for(I = 1; I <= MAX_ITER; I++){
    atomic{ f(); }
  }
}
```

```
void A::f(){
  ...
  B[thisIndex+1].recv_g_p(...);
}
```

```
entry void start_B(){
  for(I = 1; I <= MAX_ITER; I++){
    when recv_g_p(msg_g_p *p){
      atomic{ g(p->param); }
    }
  }
}
```

```
void B::g(p_type p_arg){
  // use p_arg in computation
}
```

Fig. 2. Translating an example piece of Charisma code into SDAG. Note that the `Main` object does not feature in the translated control flow specification: unlike Figure 1, evaluation of the `for` continuation criterion is done in a distributed manner, and no broadcast-barrier scheme is necessary.

Charisma, to the intra-object control flow of SDAG. We carry out this discussion in the vein of the observations made previously in § IV.

SDAG expresses local coordination and events, which by definition are pertinent only at the scope of individual objects. Therefore, we follow essentially the same projective techniques as when compiling directly to CHARM++ code. Figure 2 shows the extraction of global control statements and their inclusion in SDAG code for the appropriate object arrays. The compiler generates specifications of local control flow for the two object types in the form of SDAG methods `start_A` and `start_B`. This local coordination code is included in the SDAG interface file, denoted by the green boxes near the left edge of the figure.

The green boxes lining the right edge represent the sequential part of the Charisma code, together with generated CHARM++ code that enforces the data dependences specified in the orchestration section. Since SDAG does not facilitate the expression of global flows, we use techniques identical to those mentioned in § IV-B to enforce data dependences. Figure 2 shows the inversion of the consumed parameter index expression in `A::f()`. This method invokes the `recv_g_p` method of `B`, which eventually calls `g` on the recipient object. The presence of the `when` construct from SDAG directs the compiler to trigger execution of `g` only upon the occurrence of the corresponding message receipt event. In this example, the sequential code would be the method `g`, which consumes received data. The SDAG interface code is combined with the user code and further compiled into CHARM++.

The example here demonstrates the translation of a very simple, point-to-point communication pattern. However, Charisma allows for the expression of more complicated ones, such as multicasts, scatters and gathers. Multicasts and scatters are usually associated with syntactic groups addressing collections of objects. When translating to SDAG, for example, one must create *section proxies* to communicate with groups of objects. Further, no special form of code is required on the receiver side, apart from the necessary entry methods that are invoked on recipient objects. This is easily accomplished with the existing dependence analysis framework and is not discussed further. Data gathers, on the other hand, require that the recipient await the delivery of a number of messages (whether from every element of the producing object array, or a section thereof). The SDAG code for this kind of operation is a `when` construct enclosed within a `for` statement. The received messages are buffered and delivered to the target entry method when all of them are available.

The translation of control flow, instruction execution order and program order semantics from the global mode of Charisma to their local correspondents in SDAG merit detailed discussion. We outline translation techniques for each of the involved constructs in turn. In the following paragraphs on conditional execution and control flow translation, we distinguish between *local* and *global* predicates. The former connote predicates that only require variables local to an object; the latter may comprise variables obtained via global operations on data, such as the maximum of an error value across all objects.

A. Conditional Execution

The `if` construct in Charisma represents the execution of a list of statements contingent upon the evaluation of an associated predicate. Previously, all such predicates were evaluated on the main object. Further, these predicates could be constructed using only global parameters or variables local to the main object. In this version of the compiler, we use the SDAG `if` statement in its stead, so that these conditions *may* be evaluated in a distributed, per-object manner. This is advantageous since we can now predicate the execution of methods on local conditions, such as the index of an object. At the same time, we do not lose the expressiveness of the `if` construct, which can base statement execution on globally evaluated conditions as well. As was the case with the previous version, all variables involved in such "global" predicates (for instance, a predicate that checks the maximum error across all objects) must be relayed to the main object. The compiler currently cannot distinguish between such global and local predicates. When using global predicates, therefore, the user must include Charisma code that sends (via reductions, etc.) their various components to the main object.

B. Loop Constructs

Loops are expressed through the `for` and `while` constructs in Charisma. Here, we discuss only the more commonly used `for` construct. As stipulated by the language, `for` loops may only use local predicates to iterate over the body. For instance, in this snippet of code, `for(I = 1 to MAX_ITER)...`, `I` keeps track of the local iteration count, and the associated predicate allows execution for a specific range of values. `while` loops may include global predicates, and as before code must be included to relay the values of global parameters to the main object for centralized evaluation. As with the `if` construct, the previous compiler evaluated predicates on the main object. By translating it to the SDAG `for` statement, we allowed the distributed evaluation of the associated predicate. This obviates the implied barrier at the end of each `for` iteration, since consensus on the values of local variables needn't be maintained. This change enhances performance in iterative codes, as various iterations of a loop body are allowed to overlap to the extent possible. Of course, care must be taken to ensure the delivery of data items to their intended iterations. SDAG provides for this by labeling each message with a reference number which identifies the iteration in which it was generated. This reference number is set by our compiler, and is a function of the values of the index variables of the loop nest. Upon receipt, messages are checked for reference numbers by the existing SDAG infrastructure and delivered to the correct iteration of a loop. When dealing with loop-carried dependences, the compiler modifies these reference numbers to reflect the fact that data generation and consumption are meant to occur in different iterations.

C. Program Order

As mentioned in § II-C, Charisma provides programmers a deterministic execution semantics. Statements execute in adherence to the program order and data dependence constraints mentioned in that section. The two-phase execution strategy (mentioned in § III) of SDAG upholds the instruction order specified by the program text. Therefore, by maintaining the sequence of translated orchestration statements within the generated SDAG code, we ensure that program order is respected. Finally, Charisma programs use the `overlap` construct to relax program order in situations that do not warrant such tight controls on instruction sequence. Notice that the `overlap` construct only affects the ordering of individual objects' methods; ordering across objects is specified by data dependences. Thus, by translating the global `overlap` statement into its local version in SDAG, we do not alter semantics at all.

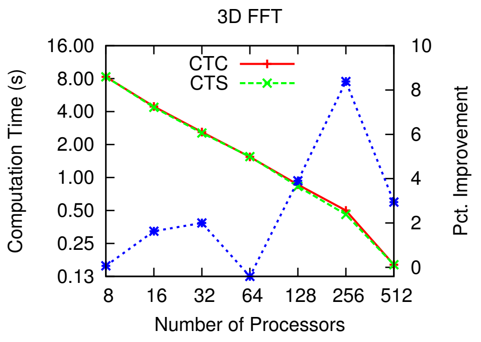


Fig. 3. Comparison of execution times of CTS and CTC in a 3D FFT program written in Charisma. The input was sized 512^3. Modest improvements in performance can be seen on larger processor counts.

VI. Performance Results

This section draws a comparison between the performance of code generated by the two versions of the Charisma compiler. As we shall see, converting a program's Charisma specification to SDAG yields certain performance benefits. This seems counterintuitive, since the SDAG version is ultimately translated to CHARM++, and direct translation, as effected in the original compiler, ought to yield quicker code. However, code translated to the intermediate representation of SDAG benefits from the efficient dynamic management of dependences, and the message-sequencing infrastructure of SDAG, which allows greater overlap between iterations of `for` loops than previously. As an added advantage, Charisma `for`-loops needn't end in barriers that return control to the main object, since the evaluation of the continuation criterion is done in a distributed manner. We discuss the performance of three staples of numerical computation: the Fast Fourier Transform, an iterative algorithm for the multiplication of matrices, and the five-point Jacobi relaxation technique. Each of these programs was written in the static data flow style of Charisma. Two versions of output code were obtained for each program; the first, low-level CHARM++ code, and the second the result of the contributions of this paper, an SDAG specification of the local sequencing of events in CHARM++ objects. Both versions (referred to as CTC, Charisma to CHARM++; and CTS, Charisma to SDAG, respectively) were then processed by the CHARM++ compiler. We detail the performance of these versions with reference to the quality of code generated by each mode of translation.

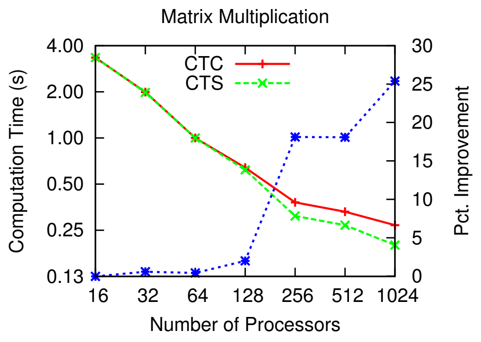


Fig. 4. Cannon's algorithm for matrix multiplication. Inputs are sized 4096×4096. The CTS version is more efficient than CTC, as shown by its better scaling profile at higher processor counts.

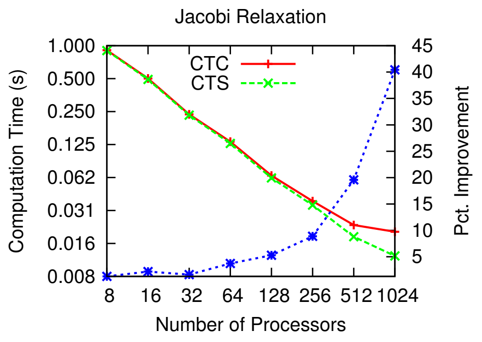


Fig. 5. Plot comparing the performance of CTS and CTC versions of a Jacobi relaxation. The input matrix was sized 4096×4096. Removing the implied barrier at the end of each iteration yields significant improvements in execution time at higher processor counts.

A. 3D FFT

Our first experiment involved a three dimensional FFT operation on input data sized 512^3. Using a transpose-based approach, we were able to make use of the corresponding one-dimensional operation provided by the FFTW library [4] to obtain good performance. Results were obtained on 8 through 512 processors of *Intrepid*. Figure 3 compares the performance of CTS and CTC. The performance of the two versions is comparable up to 64 processors. Any differences up to this point are moot given the noise likely in the collected data. Beyond this point, though, the programs start utilizing more than a single unit of processor allocation, so that communication latencies incurred by the all-to-all patterns become more marked. CTS shows mild performance gains beyond 64 processors.

B. Cannon's Algorithm

Matrix multiplication is a fairly well-studied operation with applications in a variety of scientific and numerical problems. Here, we discuss Cannon's algorithm [5], which entails the division of the result matrix C $(= A \times B)$ into tiles, each of which is computed by a different member of the worker object array. The dimensions of A, B and C are, respectively, $N \times K$, $K \times N$ and $N \times N$. The algorithm is iterative, and limits the transient memory requirements to just a tile each of A and B per computed tile of C. In each iteration, a worker uses its input A and B tiles to compute an incremental sum accumulated in its C tile. It then sends the A tile to its left neighbor, and the B tile to the object above it in the worker array.

Figure 4 presents computation times on 16 through 1024 processors for input matrices sized 4096×4096. The blue line depicts the improvement in performance of CTS over CTC. The gains are especially marked at the higher processor counts, which means that parallel overheads were reduced substantially for those runs. Indeed, it is the removal of the implied barrier at the end of each iteration of the algorithm in CTS that yields this improvement.

C. Stencil Computation

Stencil computations and other successive over-relaxation techniques find use in many scientific computing domains, chief among them the solution of linear systems. Here we discuss the Jacobi method of relaxation, which represents a system of equations as a structured grid. The algorithm partitions this grid among objects and proceeds iteratively. § III presented SDAG code for this computation and explained its structure briefly.

Figure 5 compares performance results of the CTS and CTC versions for this program on 16 through 1024 processors. Once more, as we move to a finer granularity of execution, the overheads of parallel execution become more apparent in the CTC version of the program. Since this is an iterative code, the increased overlap due to the removal of the barrier at the end of the each iteration increases its execution efficiency. The 40% improvement on 1024 processors is appreciable.

VII. Related Work

Much work has been done in the past on topics related to static data flow. Kung's pioneering [6] work on systolic arrays is especially noteworthy. O'Leary [7] presents systolic algorithms for matrix transposes and other operations. Synchronous data flow algorithms are discussed in the context of DSP applications in [8]. Rates of data production or sampling are specified a priori, a static component that finds echoes in the specification of communication patterns in Charisma. Charisma also bears similarities to Fortran M [9], where ports are connected to create channels from which point-to-point communication is generated. The SISAL language [10] ties together the notions of functional programming and streams and is representative of some of the early work done on stream programming. Work on the recently developed Concurrent Collections programming model [11] bears interesting similarities to Charisma. In particular, it engenders a separation of concerns similar to Charisma: domain experts program the application without concerning themselves with issues of parallel performance and tuning. These correspond, respectively, to the development of the sequential and parallel parts of a Charisma application.

Like Charisma and SDAG, DAGH [12] provides high-level abstractions and distributed data structures for the development of parallel applications. However, it restricts its scope to the adaptive mesh refinement technique. CC++ [13] bears some similarities to SDAG. Constructs such as *par* and *parfor* find analogues in *overlap* and *forall* from SDAG respectively. CODE [14] is a graphical language used to specify static dataflow as a visual directed graph. It uses the idea of ports in a way similar to Charisma.

Cilk [15] is a general purpose language and runtime much like CHARM++. It uses dynamic task spawning and synchronization to express multi-threaded parallelism. Its runtime system is based on work-stealing, whereas CHARM++ relies on message-driven execution. Moreover, Cilk developers focus on applications for multi-core systems. In this regard, CHARM++ offers broader portability, since efficient implementations of the runtime are available for most major architectures.

VIII. Conclusion

This paper presented a methodology for the translation of global specifications of control and data flow between parallel objects into media that allow the specification of the same program in terms of local flows. Our study was conducted in the context of a static data flow language called Charisma that engenders clear expression of global flows and improves programming productivity. We discussed the advantages of translating Charisma into SDAG, a language used to specify flow of control within CHARM++ objects. SDAG helps to coordinate actions performed on the occurrence of asynchronous message invocation events on arrays of objects. Since these are local events, SDAG essentially helps the clear specification of the local ordering of object events. We addressed some key issues in the conversion of the global expression of control flow in Charisma to the equivalent local flow specifications in SDAG. Doing this affords a number of benefits. First, since we target a fairly abstract output representation, the amount of effort required to write the compiler decreases significantly. Moreover, all the optimizations available through the SDAG framework become available to Charisma code. This includes features such as dynamic dependence checking and message sequencing to allow the overlap of neighboring iterations in iterative codes. Through the use of our methods on some staple numerical computation algorithms, we were able to demonstrate sizeable performance gains over the conventional technique of compiling to low-level CHARM++ code.

References

[1] C. Huang, "Supporting multi-paradigm parallel programming on an adaptive run-time system," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2007.

[2] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng, "Programming Petascale Applications with Charm++ and AMPI," in *Petascale Computing: Algorithms and Applications*, D. Bader, Ed. Chapman & Hall / CRC Press, 2008, pp. 421–441.

[3] L. V. Kale and M. Bhandarkar, "Structured Dagger: A Coordination Language for Message-Driven Programming," in *Proceedings of Second International Euro-Par Conference*, ser. Lecture Notes in Computer Science, vol. 1123-1124, September 1996, pp. 646–653.

[4] M. Frigo and S. G. Johnson, "Fftw: An adaptive software architecture for the fft." IEEE, 1998, pp. 1381–1384.

[5] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Montana State University, 1969.

[6] H. T. Kung, "Why systolic architectures?" *IEEE Computer*, vol. 15, no. 1, pp. 37–46, 1982.

[7] D. P. O'Leary, "Systolic arrays for matrix transpose and other reorderings," *IEEE Trans. Computers*, vol. 36, no. 1, pp. 117–122, January 1987. [Online]. Available: http://wotan.liu.edu/docis/dbl/ietrco/1987_36_1_117_SAFMTA.htm

[8] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1458143

[9] I. Foster, B. Avalani, A. Choudhary, and M. Xu, "A compilation system that integrates high performance fortran and fortran M," in *Proceedings 1994 Scalable High Performance Computing Conference*, 1994.

[10] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee, "Sisal – streams and iterations in a single assignment language, language reference manual, version 1.2," University of California - Lawrence Livermore Laboratory, Tech. Rep. TRM-146, March 1995.

[11] Z. Budimlic, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari, "Multi-core implementations of the concurrent collections programming model," in *Proceedings of the 2009 Workshop on Compilers for Parallel Computing (CPC)*, January 2009.

[12] M. Parashar, J. C. Browne, C. Edwards, and K. Klimkowski, "A common data management infrastructure for adaptive algorithms for pde solutions," in *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM, 1997, pp. 1–22.

[13] K. M. Chandy and C. Kesselman, *CC++: A declarative concurrent object-oriented programming notation*. MIT Press, 1993.

[14] J. C. Browne, S. I. Hyder, J. Dongarra, K. Moore, and P. Newton, "Visual programming and debugging for parallel computing," *IEEE Parallel Distrib. Technol.*, vol. 3, no. 1, pp. 75–83, 1995.

[15] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Journal of Parallel and Distributed Computing*, 1995, pp. 207–216.