Efficient Implementation of High Performance Fortran via Adaptive Scheduling–An Overview *

Edward A. Kornkven
Department of Computer Science
University Of Illinois
Urbana, IL 61801
kornkven@cs.uiuc.edu

Laxmikant V. Kalé
Department of Computer Science
University Of Illinois
Urbana, IL 61801
kale@cs.uiuc.edu

We have been developing a compiler for a subset of High Performance Fortran. We have shown that generating message-driven code provides an opportunity for improved efficiency in the presence of communication. By utilizing a notation called Dagger, we are able to schedule work efficiently without relying on complicated and unreliable compile-time approaches. This paper gives an overview of our approach and reports on the project's status.

1 Introduction

Some of the factors which might negatively affect performance in a data parallel program are long communication latencies, synchronization, and load imbalance. These factors are related in that they hinder performance by causing processor idle times, and their effects might be diminished or eliminated by reordering the statements of the program. In the case of communication latencies, since remote data accesses take longer than local ones, it is desirable to mask the latencies of the communication by overlapping those accesses with other computations. Synchronization may often be relaxed–for example, the global program counter of the data parallel virtual machine need not be physically realized as long as the semantics of the user program are preserved. In the case of load balancing, the overall execution time of a program between two global synchronization points is exactly the time required by the processor that finishes last. Hence, in order for the computation to be speeded up, some of the work done by that processor must be transferred to other processors. One might expect load imbalances to be a relatively unimportant factor in the performance of "regular", data parallel computations since data is usually distributed across processors more-or-less evenly. Even for these problems, however, uneven workload distributions across processors may exist because some processors may carry more elements than others, and some elements may require a significantly greater amount of processing. In this research, we handle these problems via a novel approach to scheduling the instructions of the program. By "scheduling" instructions, we mean specifying the order in which they are executed. This includes determining when processors must synchronize (to wait for data that is necessary for a later processing step) and choosing among alternatives when more than one instruction could execute. In the simplest implementation, the compiler could generate code that matches exactly the order of instructions specified by the source program. But, as we have seen already, such an implementation could miss opportunities to improve efficiency that are afforded by the MIMD architecture. Another approach is to do compile-time analysis to enable instructions to be reordered into a schedule that better mitigates communication latency and load imbalance factors. However such static methods may not be feasible due to a lack of information at compile time.

We believe that an instruction schedule will be most effective if it is expressed as a partial order that is executed in a data-driven fashion so that it is adaptable to run-time conditions as they unfold. At compile time, it can be difficult or impossible to identify code that will lead to unnecessary idle times. For instance, the cost of executing a particular program statement may depend on run-time inputs to the program–schedules based on compile-time analysis alone can only "guess" at what run-time conditions

*This research was supported in part by National Science Foundation grants CCR-90-07195 and CCR-91-06608.

will be. Program instructions must be reordered to avoid unnecessary idle times, but in general, the number of statements needed (and available) to reorder is influenced by run-time conditions and cannot be completely predicted statically. For instance, at compile time we may not know what the run-time load of each processor will be, exactly which instructions will execute for each processor, what the problem size will be, perhaps the number of processors that will be used during execution, or perhaps even what kind of processors they will be.

In this paper, we will show how we approach this scheduling problem by generating adaptive schedules at compile time that enforce all the control and data dependences correctly (so that necessary orderings are preserved), while allowing the run-time system to reorder processing of different messages (and associated subcomputations) depending on the arrival order of messages at run-time (and thus *adapt* to run-time conditions). We have implemented this technique in a compiler for a subset of High Performance Fortran (HPF) [6] called DP. DP supports operations on entire arrays such as arithmetic operations and assignment, an assortment of array- and scalar-oriented intrinsics, and HPF's control structures: indexed DO, DO WHILE, FORALL and if statements. Arrays are distributed among processors using HPF's array distribution directives, and each processor computes its own portion of an array. The schedule and computations are specified using a language called Dagger, which is part of the Charm parallel programming system [7]. Charm is an explicitly parallel language with a message-driven execution model. Since Charm is machine independent and has already been ported to numerous shared-memory and nonshared-memory parallel machines, the code produced by the compiler is machine independent by default. The analysis required to generate good schedules is not particularly difficult or expensive, leading to simpler code generation. Furthermore, this approach is orthogonal to other kinds of optimizations, enabling a compiler to take advantage of many other code improvements such as optimization of data mapping, more detailed dependence analysis, and communication combining.

2 Our Approach to Scheduling

We discussed in Section 1 the potential benefits of constructing an instruction schedule that is adaptable to run-time conditions. What this means in practical terms is that the final order of execution must be determined at run-time. However, there are many constraints on that order–namely the control and data dependences in the source program–and these can be determined (though not satisfied) at compile time.

We wish to find a minimal set of constraints on the ordering of basic blocks (in our implementation, each block is one or more statements in our intermediate language). Using some relatively straightforward compile-time analysis, we construct a graph of basic blocks in which the edges of the graph capture dependences between blocks. In simplified terms, when a block's predecessors in the graph have completed, the block is labelled as ready for execution by the run-time system (which also selects and initiates ready blocks). Execution begins with blocks which have no predecessors and continues in a *macro-dataflow* fashion [13] until execution is complete.

In summary, our approach depends on three components:

1. A message-driven run-time substrate to provide the ability to execute a multi-threaded schedule that is adaptable to run-time conditions.
2. The ability to express the computation as a partial ordering of sub-computations.
3. Analysis by the compiler to obtain the partial ordering of the instructions to be scheduled.

The first of these components is realized by the Charm parallel programming system, the second by the Dagger notation that runs on top of Charm. The third is provided by the DP compiler which emits Dagger code as its target language. Charm and Dagger will be briefly discussed in the Section 2.1. An overview of compilation of DP to Dagger is presented in Section 2.2. sections.

2.1 Charm and Dagger

Charm supports the dynamic creation, manipulation, and scheduling of small tasks called *chares*. Chares may create other chares or send messages to entry points of existing chares, enabling those chares to be scheduled for execution. Entry points are designed to execute for a relatively short time to yield a "medium-grained" computation. A special kind of chare, the *branch office* chare (BOC), is automatically replicated on each processor. BOCs are useful for describing a computation that is similar on all processors–such as the array processing in DP. Each processor executes its own instance of a BOC on its own data resulting in SPMD-style processing.

Charm supports an *asynchronous*, *message-driven*[1]

[1]Note that *message-driven* computation is very different from *message-passing*.

programming style in an explicitly parallel C-like language. Chares begin executing at a certain entry point in response to the receipt of a message directed to that entry point. There are no *receive* statements in the language. A chare that sends a message continues executing the entire entry point and then suspends, waiting for another message. In the meantime, other messages that have arrived are de-queued and execution continues, possibly in another chare.

Dagger [5] runs on top of Charm and is a notation for expressing computations that can be represented by directed graphs. Dagger facilitates the expression of the local synchronization of threads of execution by specifying the condition under which a thread executes in terms of predecessor threads that must complete first. In the DP compiler, we use Dagger to schedule statements and messages of an executing program–Dagger is the target language of our compiler. At compile time, the conditions under which a statement may execute are described but the order in which they will actually execute depends on which of those conditions are met first during a particular execution of the program. Using Charm constructs, Dagger can in effect "execute" a dependence graph. This results in a dataflow-flavored approach which enables statements to be executed at their earliest opportunity.

Dagger allows one to define a special form of chare in which the code is segmented into `when` blocks. A particular `when` block is tagged with dependences that must be satisfied in order to enable the `when` block to be initiated for execution by the Charm run-time system. These dependences assume one of two main forms in Dagger:

1. An *entry*–the target of a message that will be sent to the chare by some PE, or
2. A *condition variable*–a special variable that Dagger watches.

A function called `Ready` is used to set the condition variables, which are initialized to be "not ready". Syntactically, `when` blocks have the form

`when` $cv_1, cv_2, \ldots, cv_m, e_1, e_2, \ldots, e_n$:
  { *code to be executed* }

When all condition variables cv_i are ready, and for each controlling entry e_j an `expect` statement has been executed and a message directed at e_j has been received, the `when` block is ready to execute. When it actually executes depends on which ready `when` block Dagger next selects for execution.

For example, given the following DP code fragment,

```
asum = SUM(A)
bsum = SUM(B)
C = A * asum + bsum
```

the Dagger `when` blocks that might correspond to these statements are as follows:

```
when A_computed: {
  Compute sum of elements of A on this PE
  Send local sum to reduction library
  /* Library broadcasts result in amsg */
  expect(amsg); }
when amsg: {
  asum = amsg->result;
  ready(asum_computed); }
when B_computed: {
  Compute sum of elements of B on this PE
  Send local sum to reduction library
  expect(bmsg); }
when bmsg: {
  bsum = bmsg->result;
  ready(bsum_computed); }
when asum_computed, bsum_computed: {
  Do C=A*asum+bsum on local elements
  ready(C_computed); }
```

Note that either `SUM(A)` or `SUM(B)` can begin first, depending on whether `A` or `B` is ready first.

2.2 Compile-Time Analysis

To illustrate the actions a compiler might take in translating DP to Dagger, consider the simple DP code fragment from the last section. As stated previously, we wish to enable statements to be reordered at run time. We will first create a data dependence graph (DDG) for the program to assist in our analysis. In the current compiler, we have chosen to keep the dependence analysis simple. For instance, and we don't compute dependences (or the absence of dependences) revealed by analysis of array subscripts. Future versions of our compiler would benefit from more in-depth dependence analysis.

Our DDG has two kinds of nodes. A computation node will translate to a `when` block in the Dagger program. A `when` block is labelled by condition variables and entries corresponding to other DDG nodes upon which the node depends. Message nodes in the graph represent points at which messages are received and have no corresponding computations. Message nodes are of special interest because they indicate a dependence on a run-time event which cannot be predicted by the compiler (namely, the receipt of a message).

Analysis of Branches and Loops

In the presence of loops, analysis is complicated by the fact that, due to branching, it is no longer possible at compile time to determine precisely which statements will generate the values used at run time by other statements. For example, suppose we have a Jacobi iteration code fragment as shown below.

```
Anew = init_val
DO WHILE (not_done(Anew,A,epsilon))
  A = Anew
  Anew = (1/(2+2*c)) * (CSHIFT(A,1,-1) +
    CSHIFT(A,1,1) + c*(CSHIFT(A,2,-1) +
    CSHIFT(A,2,1)) - W)
END DO
D = Anew - A
```

A and $Anew$ are defined inside the loop. First, the statement which computes D must be enabled by *either* the defining statement in the last iteration of the loop or the statement that defines the variable if the loop is not executed (i.e., its defining statement before the loop). In terms of Dagger, the statement $D = Anew - A$ will appear in a `when` block that depends on condition variables that trigger the `when` block after either (a) the definitions of A and $Anew$ in the loop, or (b) the previous definitions of A and $Anew$ if the loop is not taken. Second, if the loop is taken (i.e. in case (a)), the A and $Anew$ definitions inside the loop must not trigger the $D = Anew - A$ statement until after the last iteration of the loop. Also, suppose that for some arbitrary processor P, a neighboring processor Q sends a message to P (say, its left boundary for a left CSHIFT). It could happen that before P processes this message, it sends its right boundary to Q (for the right CSHIFT), Q receives and processes all of its messages, and begins the next iteration. Q then sends another message to P before P has processed the first. P must be able to match each message with the proper iteration.

A Dagger program is completely data-driven–`when` blocks are activated under certain conditions which correspond to the computation of data values or the arrival of messages. So we must construct a looping mechanism which uses the correct values for each iteration using this framework. We must also have a way for multiple independent execution paths to be coordinated by a single control mechanism (i.e. the loop condition) while honoring the loop-carried dependences. These requirements are satisfied by locally synchronizing all threads through the loop after each loop iteration.[2] This nullifies loop-carried dependences but may miss some opportunities for overlap across iterations. This is sufficient to coordinate control within a processor. To deal with the problem of multiple messages arriving from different loop iterations, we employ a Dagger mechanism called *reference numbers.*

A reference number is a tag that may be attached to a condition variable or message to enable Dagger to group a set of condition variables and messages together. This enables Dagger to distinguish between different instances of a computation (e.g. different iterations of a loop). To use a `when` block that depends on a set of such messages and condition variables, condition variables and entries are declared with the `MATCH` option which tells Dagger to match reference numbers when checking for fulfillment of a `when` block's dependences. A `when` block is not activated for execution until (a) a `Ready` statement (using a reference number) is executed for each of its condition variables cv_i, (b) for each of its entries e_i an `expect` for the same reference number has been executed, and (c) for each of its entries e_i, a corresponding message with the same reference number has been received. The reader is referred to [5] for more details. The details of the design of the Dagger code generation algorithm may be found in [8].

3 Performance

Using a synthetic program consisting of eight WHERE statements separated by SUM calls, we conducted some experiments to determine the potential effectiveness of the scheduling method described here. The characteristics of this example that make it useful for our purposes are the following:

1. The eight WHERE statements are part of two main "threads" of execution.
2. The four WHERE statements of each thread must execute in succession.
3. Either thread is available to be started after some initial computations are completed.
4. The amount of work performed on a processor in each WHERE statement differs for each processor.

In this program, the workload is unbalanced throughout the computation on nearly every processor. That imbalance is not going to be obvious to a

[2]Notice that we are *not* synchronizing across all the processors. We are only coordinating the `when` blocks within a loop, on each processor separately.

compiler because it is a function of the *data* of the program as well as the statements of the program. However, in our compiler, that fact is not critical because the schedule is essentially created at run time according to run-time conditions, one of those being the program data.

We compared the multithreaded code generated by the DP compiler to a single-threaded version of the same program. The two versions of the program were run on a 32-processor CM-5 and analyzed using *Projections* [14], a performance analysis tool for Charm. Using this tool, we are able to clearly see the performance characteristics of the two versions of the program by looking at overall system utilization. The single-threaded version (Figure 1), has eight prominent spikes corresponding to the eight WHERE statements in the program, with relatively long idle times between each one. These idle times are due to the load imbalances–as time progresses through a WHERE statement, fewer processors are participating in the computation and overall system utilization tails off slowly over a relatively long period of time as processors wait to synchronize. In the multithreaded version on the other hand (Figure 2), the eight spikes become four plateaus, corresponding to pairs of WHERE statements that are scheduled together. Real programs won't often fit our approach so perfectly, but the principle is illustrated well by this example.

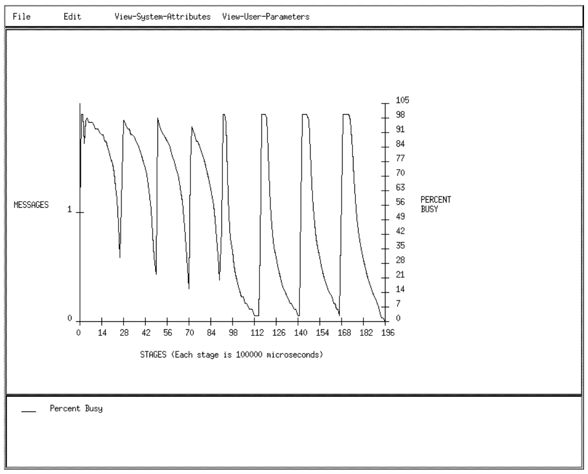

Figure 1: Single-threaded aggregate utilization

Charm Overhead

One obvious question that arises when comparing any dynamic approach to static alternatives concerns the run-time costs involved. In the context of this work, there is run-time overhead involved every time

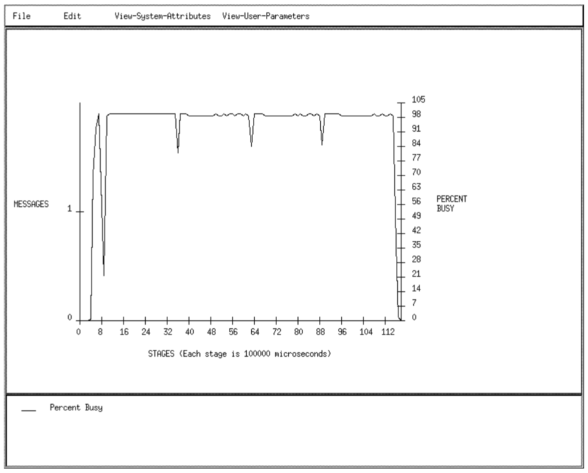

Figure 2: Multi-threaded aggregate utilization

control is relinquished to the Charm run time system. This will occur approximately once per `when` block. Each `when` block will contain at least one DP statement, often several. So we could estimate the number of "calls" to Charm to be roughly equal to the number of DP statements. Obviously then, the amount of user work done per statement will determine the extent to which Charm overhead is amortized by the user computations. [3]

4 Project Status

Our DP compiler currently generates Dagger code for an HPF subset language for all array distributions defined by HPF. We plan to use DP to handle the regular, array-based computations in Charm applications. The Charm language itself is particularly well-suited to irregular computations. DP subroutines will be callable from Charm, Dagger, and other languages that we provide, giving the user the flexibility to choose the appropriate tool for either regular or irregular computations. Similarly, Charm routines will be callable from DP. To that end, we have defined a data interface between Charm and DP, and have developed some library routines that are callable from either the DP environment or Charm. These include array distribution and redistribution, reductions, array and array section assignments, and dot product.

[3]We timed Dagger to get a concrete idea of the cost of its overhead. The overhead for each `when` block was measured to be about 27 microseconds per iteration on a Sun Sparcstation 1. We can expect the computation in an array-based application to overwhelm this overhead.

The method we describe leaves open many optimizations that can be performed, including those to reduce the scheduling overhead incurred. These include elimination of redundant dependences in certain `when` blocks, a reduction in the number of condition variables used, etc. We plan to explore these opportunities as well as extend and improve the DP compiler and further evaluate its performance. Details about DP and its implementation will be found in [9].

5 Related Research

Numerous efforts have been undertaken in recent years to compile languages based on data parallelism to MIMD machines. Among these are numerous Fortran variants [4, 15, 16, 2, 11]. Many of these efforts explicitly address the problem of generating efficient communication. What distinguishes our work is that the order in which tasks are performed is determined at run-time according to our adaptive schedule. We know of no other machine-independent run-time system that offers the asynchronous, message-driven virtual machine to which we compile.

Some examples of recent Fortran projects in which efforts are made to overlap communication and computation and/or improve load balance include Fortran D [15], Vienna Fortran [16], Fortran 90D/HPF [2], Data Parallel Fortran [4], and the Crystallizing Fortran Project [11]. To our knowledge, all of these efforts rely on statically scheduling program tasks.

Finally, we note that previous work on dependence analysis [10, 1, 12] and computing static single assignment forms [3] has been helpful to our analysis.

References

[1] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[2] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers. In *Proc. of Supercomputing '93*, pages 351–360, Nov. 1993.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, Oct. 1991.

[4] Pablo M. Elustondo, Lelia A. Vazquez, Osvaldo J. Nestares, Julio Sánchez Avalos, Guillermo A. Alvarez, Ching-Tien Ho, and Jorge L.C. Sanz. Data parallel Fortran. In *Proc. of Frontiers '92*, pages 21–28, Oct. 1992.

[5] A. Gursoy and L.V. Kale. Dagger: Combining the benefits of synchronous and asynchronous communication styles. In *Proc. of IPPS '94*, pages 590–596, Apr. 1994.

[6] HPF Forum. *High Performance Fortran Language Specification*, 1.0 edition, Jan. 1993.

[7] L.V. Kale. The Chare Kernel parallel programming language and system. In *Proc. of ICPP '90*, volume II, pages 17–25, Aug. 1990.

[8] E.A. Kornkven. Dynamic adaptive scheduling in an implementation of a data parallel language. Technical Report 92-10, Parallel Programming Lab., Dept. of Computer Science, University of Illinois, Oct. 1992.

[9] E.A. Kornkven. *Exploiting Message-Driven Execution to Compile Data Parallel Programs.* PhD thesis, University of Illinois, 1994.

[10] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proc. of POPL '81*, pages 207–218, Jan. 1981.

[11] Jingke Li and Marina Chen. Compiling communication-efficient programs for massively parallel machines. *TPDS*, 2(3):361–375, July 1991.

[12] William Pugh. A practical algorithm for exact dependence analysis. *CACM*, 35(8):102–114, Aug. 1992.

[13] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* The MIT Press, 1989.

[14] A.B. Sinha. *Performance Analysis of Message Driven Programs.* PhD thesis, Dept. of Computer Science, University of Illinois, 1994.

[15] C.W. Tseng, S. Hiranandani, and K. Kennedy. Preliminary experiences with the Fortran D compiler. In *Proc. of Supercomputing '93*, pages 338–350, Nov. 1993.

[16] H. Zima, B. Chapman, H. Moritsch, and P. Mehrotra. Dynamic data distribution in Vienna Fortran. In *Proc. of Supercomputing '93*, pages 284–293, Nov. 1993.