Compiling Task and Data Parallel Programs for iWarp

(Extended Abstract)

T. Gross, S. Hinrichs, G. Lueh, D. O'Hallaron, J. Stichnoth, and J. Subhlok

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

1 Introduction

Compiling for private memory parallel machines is a difficult problem and an active area of research. There are many reasons why compiling for such machines is difficult: First, there is no single programming model. Second, there are a variety of compilation strategies, and no one strategy is best for all situations. Third, parallel architectures are still evolving, providing new opportunities and new challenges.

Two widely used programming models for private memory machines are data parallelism [7, 10, 17, 16, 9] and task parallelism [4, 12, 11, 13]. Data parallelism is typically expressed as operations on arrays of data, and is especially useful when the size of the arrays can be scaled to fit the size of the parallel machine. Task parallelism comes in several flavors, including independent parallel loops and sequential processes with arbitrary communication patterns between processes.

Many applications exhibit data and task parallelism [5], and for some important applications it is essential to exploit both. E.g., a real time signal processing application can typically be expressed as a pipelined collection of data parallel tasks. One way to compile such an application is to ignore the task parallelism, and to execute each data parallel task in sequence, using all processors for each task. However, this can lead to poor processor utilization, since the number of sensors and the sampling rate (and thus the size of the data) are limited by hard physical constraints. This kind of phenomenon was encountered in [14], where it was shown that only up to 8 processors could be used efficiently in a purely data parallel spectral detection system. The solution is to exploit both styles of parallelism by allocating a subset of processors to each data parallel task.

There are a number of different approaches to compiling for private memory parallel machines. E.g., for some computations, a compiler can gather sufficient information so that computation and communication can be scheduled on a fine-grain basis [15]. For other computations, decoupling computation and communication is the only option, and this is the approach taken by a number of compilers that rely on message passing [17, 16]. For yet other computations (e.g., irregular sparse matrix operations as they arise in finite element methods), indirect accesses to memory make compile-time analysis difficult. In those cases it is essential to allow the programmer to give directives to the compiler.

In this paper, we describe how these problems are addressed in a prototype Fortran compiler that has been implemented at Carnegie Mellon. The target machine for the prototype is the iWarp system, a private memory parallel computer developed jointly by Intel Corp. and Carnegie Mellon [1]. Section 2 describes the compiler model. Section 3 describes the implementation for the iWarp system.

2 Compiler model

The compiler model is based on two observations: First, the abstract machine that the compiler targets must be realistic enough to model the features of modern parallel architectures. Second, we recognize that different code generation strategies require binding of machine resources at different times. E.g., if the compiler produces code that tightly couples communication and computation, then dedicated communication channels must be made available between participating processors. If the owner-computes rule is used to partition data and computation, then message passing may be used to transfer data, and binding decisions can be delayed until runtime.

2.1 Abstract machine

For a system with p physical processors, the abstract machine is a collection of p *logical processors*, each with associated private memory, that communicate with other processors only via *connections*. A connection $c(i, j)$ is a dynamic communication channel between logical processors i and j. Items put by the sender i into the channel are received by j in order. To create a connection $c(i, j)$, logical processors i and j each invoke the function *c = make(i,j)*, which returns a handle to the connection. Once the connection is created, i transfers a stream of words to j by executing send commands of the form *send(c, addr, n)*, with *addr* the starting address and *n* the number of words to be transferred. Similarly, j receives a stream of m words over the connection by executing receive commands of the form *result = receive(c, m)*. The number of *sends* and *receives* do not have to match; the only requirement is that the total number of words sent and received over the life of the connection must be the same. The connection is destroyed by executing *break(i,j)* on both i and j. The sender and receiver do not have to execute *make()* and *break()* simultaneously. The only requirement is that no send operation is executed before the sender has executed *make()*, and all data transferred over a connection must have been received before the receiver can *break()* the connection.

Supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972–90–C–0035 issued by DARPA/CMO to Carnegie Mellon University, and in part by the Air Force Office of Scientific Research under Contract F49620–92–J–0131.

Connections serve several purposes. First, they represent the reservation of communication resources. Depending on the target machine, resources might be bandwidth, guaranteed bandwidth, network ports, or communication queues. They may be implemented in a variety of ways: with dedicated hardware resources, with software simulation, or some combination of these. The abstract machine leaves such implementation details unspecified. Second, connections capture the temporal locality that exists between processors; a processor that communicates with another processor once is likely to do so again in the near future. A connection-based machine model allows for some optimizations that are not available to machine models that base all communication on short-lived messages.

2.2 Program representation

The internal program representation used by the compiler is a graph called a *uniform task graph (UTG)*. Each node in a UTG is a computation, which is a thread of control with associated private memory. Each edge represents a data movement or a control dependence between two tasks. A task waits for its input data to be ready and for the control precedence constraints to be satisfied, performs its computation, outputs its results to other tasks, and then terminates. A task runs on a single logical processor. Multiple tasks may be assigned to the same logical processor.

The structure of the UTG is hierarchical, and is similar to the representations used by other researchers [3, 6]. The UTG is a compact representation of the computation to be performed, and captures sufficient information for most of the decisions that a parallelizing compiler has to make.

2.3 Compilation approach

The idea is to capture all levels of parallelism from fine-grain to coarse-grain in a single representation. Even though we use different strategies for exploiting parallelism at different levels, the existence of a single framework makes it possible to reason about different levels of parallelism in a systematic way. In particular, when an application consists of a small number of coarse-grain tasks, the compiler uses information about parallelism and execution times of individual tasks to allocate processors to tasks.

The main steps in the compilation process are outlined in Figure 1. The first phase of the compiler creates a UTG for the program using data flow analysis and data dependence analysis.

Next comes a sequence of steps that manipulate the UTG. *Message blocking* attempts to move inter-task communication out of loops. *Data alignment*, which determines the data items that should be on the same processor, and *connection prioritization*, which assigns weights to the edges of the UTG, compute auxiliary information for later use by the partitioning phase. This phase makes no assumptions about the resources in the target machine.

The next phase, code and data partitioning, maps the computations and data associated with the nodes of the UTG to logical processors (one logical processor for each physical processor), and then generates abstract machine code for each logical processor. There are a number of different strategies that can be applied:

1. *Owner-computes data parallelism.* Data is distributed across processors. Each data item is owned by one processor, which makes all updates to that data item.
2. *Coarse-grain task parallelism.* Tasks are either independent parallel loop iterations or sequential processes whose communication pattern is either given by the user or derived by the compiler. Each task is allocated to a set of processors in the target machine.
3. *Systolic parallelism.* Communication and computation are integrated in a tightly coupled, fine-grained manner.

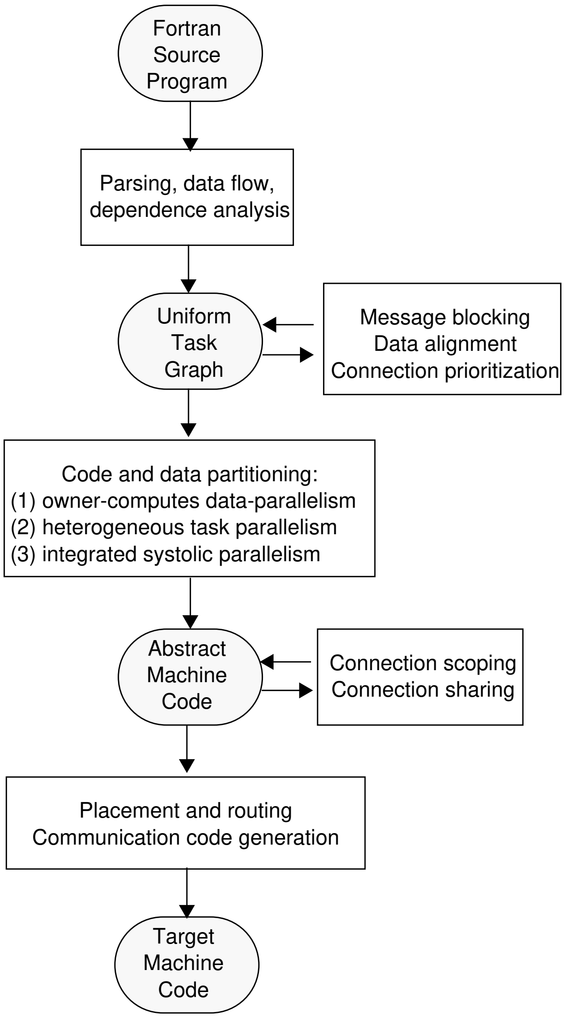

Figure 1: Compiler model

The particular partitioning strategy chosen is determined by user hints and compiler analysis. Depending on the strategy chosen, some decisions on resource usage in the target machine may become necessary. For example, a loop that uses fine-grain systolic communication will need dedicated communication resources to get good performance. The result of this phase is code for each of the logical processors in the abstract machine.

Next come transformations on the abstract machine code that arise from the existence of connections in the model. In general, the abstract machine program produced by the previous phase will contain repeated sequences of code to (1) make a connection, (2) transfer data, and then (3) break the connection. *Connection scoping* attempts to increase the time between *make()* and *break()* statements, thus amortizing the runtime cost of creating and destroying connections. Another transformation, *connection sharing*, attempts to identify cases where it is possible for different connections to share physical resources. In each case, we are making decisions about resource usage in the target machine. However, none of these decisions is binding, and any such decision can be undone by later phases.

The final phase of the compiler translates the abstract machine code into target machine code. The major decisions in this phase are deciding the layout of logical processors onto the physical processors, and determining the implementation of connections. These decisions are made from information collected in earlier phases, and from the structure of the abstract machine program. On iWarp, connections can be implemented with dedicated resources (i.e., *pathways*), or with conventional message passing. Pathways are more efficient but consume limited hardware resources. The approach is to implement the important connections as pathways, and to implement the rest using message passing.

3 iWarp implementation

We are developing a compiler for the iWarp machine that exploits data and task parallelism in Fortran programs. The input language is Fortran 77 with array extensions of Fortran 90. The compiler generates a Fortran 77 program for a collection of logical processors, which are then mapped one-to-one to the physical processors of the iWarp array.

The compiler relies on both analysis and user directives. Data parallel directives allow users to distribute and align arrays, and to identify parallel loops. They are similar to the directives in Fortran D [9] and the evolving HPF Fortran standard. Task parallel directives designate parts of the program that should be compiled as independent tasks, describe their data dependence relationship, and optionally give relative execution time information for load balancing. All directives are special comments, thus any program can be compiled and run using a standard Fortran 90 compiler.

The compiler hierarchically partitions the program using available data parallelism and task parallelism. Figure 2 depicts the steps of Figure 1 as used in compiling a small Fortran program with four tasks for iWarp. In general, the number of tasks can be large and even unknown. A data parallelism phase analyzes the array statements and parallel loops, and generates parallel code for individual procedures. A task parallelism phase analyzes user directives that specify which procedure calls should be executed as parallel tasks, and introduces a layer of the uniform task graph that shows the dependence and communication relationship between tasks. Sets of tasks are coalesced to form a *module graph*. A module is a unit for placement on the processor array. The modules are placed on the processor array and the communication channels are allocated using an existing iWarp tool.

3.1 Example program

The following example is a 2D FFT that captures the structure of many real-time signal processing applications:

```
do i = 1,m
   call rowffts(A,n)
   call colffts(A,n)
   call consume(A,n)
enddo
```

This loop performs a sequence of m $n \times n$ 2D FFT's. Procedure *rowffts* applies a 1D FFT to each row of matrix A; *colffts* applies a 1D FFT to each column of matrix A; *consume* manipulates the resulting matrix.

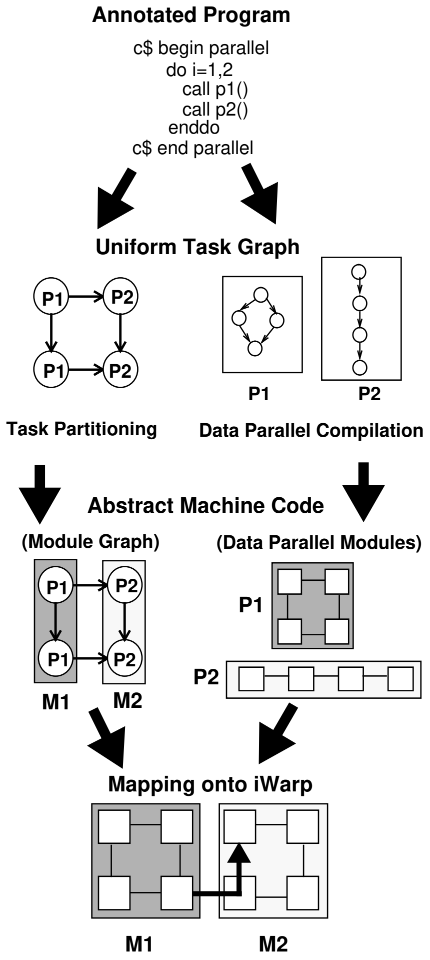

Figure 2: Example compilation for iWarp.

To compile the program in data parallel mode, we use directives to specify alignment and distribution and to indicate which program loops are parallel loops. In this example, we distribute the two dimensional array A row-wise in the procedure *rowffts* and column-wise in the procedure *colffts*. In each iteration of the outer loop, all processors compute the row FFT's, then transpose the data, then compute the column FFT's, and finally consume the data.

We can also compile this program by assigning different parts of the processor array to different procedures, and pipelining the computation. Depending on the problem parameters like the sizes of the matrices, this can be the most efficient way of executing this program. We can generate such a parallel program using directives for task parallelism:

```
C$      begin parallel
        do i = 1,m
           call rowffts(A,n)
C$         output: A
           call colffts(A,n)
C$         input: A, output: A
           call consume(A,n)
C$         input: A
        enddo
C$      end parallel
```

The compiler creates four data parallel modules, one for each subroutine call and one for the main program. The layout of the modules and the communication between them is shown in Figure 3. The communication channels form a pipeline between the three procedures that are compiled as tasks. An additional channel is created from the last procedure in the pipeline to the main program to store the final result. Currently user directives are used to guide layout of the modules. We are experimenting with strategies to automatically determine the allocation and mapping of processors to tasks.

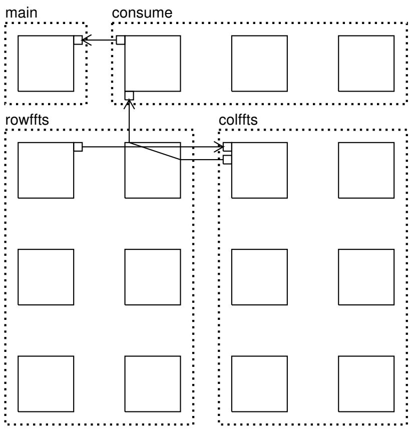

Figure 3: Layout and communication of task modules

3.2 Selecting and optimizing communication

Generation of efficient communication is critical to obtaining good performance on private memory machines. Two important communication optimizations for iWarp are: utilizing long-lived connections and reducing synchronizations.

Any connection that transfers one message can be implemented with a message passing library, which uses a fixed amount of communication resources. Connections that transfer several messages may be better implemented by having separate longer-lived communication resources assigned to them. In many cases, a dynamic programming algorithm can be used to optimally assign the limited number of dedicated pathways to the potential long-lived connections [8].

To guarantee message ordering on a MIMD machine, it is necessary, in general, for the processors involved in a *make()* to be synchronized. However, in many cases the synchronization can be safely eliminated. For example, no synchronization is needed between two *make()* calls when the receiving processors are distinct, because each processor will receive at most one message; and no synchronization is needed when the sending and receiving processors are the same as in the previous *make()* call, because the communication system on iWarp ensures that messages sent between the same pair of processors will arrive in the order sent. These and other rules can be used to create a data flow problem to remove unnecessary synchronization over the entire program.

4 Concluding remarks

We have presented a model of compilation for a private memory machine with support for connections between processors, e.g., the iWarp system. This machine model captures processor locality in a private memory machine and exposes many communication optimizations to the compiler, e.g., the management of long-lived connections.

Our compiler for iWarp is based on a uniform representation of parallelism, and this allows it to handle both task and data parallelism in a single application program. Our experience confirms that an integrated approach to exploiting different styles of parallelism is necessary for achieving good performance for many applications on modern private memory parallel machines.

References

[1] S. Borkar et. al. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing '88*, Kissimmee, FL, November 1988.

[2] N. Carriero and D. Gelernter. Applications experience with Linda. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, pages 173–187, New Haven, CT, July 1988.

[3] R. Cytron, J. Ferrante, and V. Sarkar. Experiences using control dependence in PTRAN. In *Languages and Compilers for Parallel Computing*, August 1989.

[4] J. Dongarra and D. Sorensen. A portable environment for developing parallel Fortran programs. *Computer Computing*, 5:175–186, 1987.

[5] G. Fox. The architecture of problems and portable parallel software systems. Technical Report CRPC-TR91172, Northeast Parallel Architectures Center, 1991.

[6] M. Girkar and C. C. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, March 1992.

[7] D. W. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[8] S. Hinrichs. Optimizing program communication for private memory machines. In *Languages and Compilers for Parallel Computing*, 1992.

[9] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[10] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic domain decomposition in BLAZE. In S. K. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, pages 521–524, August 1987.

[11] M. Lam and M. Rinard. Coarse-grain parallel programming in Jade. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[12] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[13] D. R. O'Hallaron. The ASSIGN parallel program generator. In *Proceedings of the 6th Distributed Memory Computing Conference*, pages 178–185, Portland, OR, April 1991.

[14] H. W. Printz. *Automatic Mapping of Large Signal Processing Systems to a Parallel Machine*. PhD thesis, Carnegie Mellon, May 1991.

[15] H. B. Ribas. Automatic generation of systolic programs from nested loops. Technical Report CMU-CS-90-143, Department of Computer Science, Carnegie-Mellon University, June 1990.

[16] P. S. Tseng. A parallelizing compiler for distributed memory parallel computers. Technical Report CMU-CS-89-148, Department of Computer Science, Carnegie-Mellon University, May 1989.

[17] H. Zima, H-J. Bast, and M. Gerndt. *SUPERB: A Tool For Semi-automatic MIMD/SIMD Parallelization*, volume 6 of *Parallel Computing*, pages 1–18. North-Holland, 1988.