Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, May 1993, San Diego, CA

Exploiting Task and Data Parallelism on a Multicomputer

Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

For many applications, achieving good performance on a private memory parallel computer requires exploiting data parallelism as well as task parallelism. Depending on the size of the input data set and the number of nodes (i.e., processors), different tradeoffs between task and data parallelism are appropriate for a parallel system. Most existing compilers focus on only one of data parallelism and task parallelism. Therefore, to achieve the desired results, the programmer must separately program the data and task parallelism. We have taken a unified approach to exploiting both kinds of parallelism in a single framework with an existing language. This approach eases the task of programming and exposes the tradeoffs between data and task parallelism to the compiler. We have implemented a parallelizing Fortran compiler for the iWarp system based on this approach. We discuss the design of our compiler, and present performance results to validate our approach.

1 Introduction

Many applications can be naturally expressed as collections of parallel or pipelined tasks, with coarse grained parallelism among the tasks, and finer grained parallelism within each task [7]. For example, a sonar spectral detection system consists of a sequence of parallel pipelines, where each pipeline consists of a sequence of downsampling time domain filters, followed by an FFT, thresholding, and other postprocessing operators [16].

Two widely used styles of parallelism for private memory multicomputers are data parallelism [9, 11, 20, 19, 10] and task parallelism [6, 13, 12, 14]. Data parallelism is typically expressed as a single thread of control operating on data sets distributed over all nodes. It is especially useful when the size of the data sets can be scaled to fit the size of the parallel machine. Task parallelism is typically expressed as a collection of sequential processes with explicit communication between them.

For some applications it is essential to exploit both styles of parallelism within the same program. In particular, for signal processing applications, physical considerations often make it impossible to arbitrarily increase the size of the data sets. Thus, it may be impossible for a single data parallel thread of control to use all nodes effectively. For example, in a signal processing application the number of input sensors can be limited by size, weight, power, and cost considerations. Further, the number of frequencies of interest is constrained by the sampling rate and the physical characteristics of the signal being analyzed. The result is that, as the number of nodes in modern multicomputers increases, it becomes increasingly difficult to use all nodes effectively on a single task. This phenomenon was encountered in [15], where it was shown analytically that at most 8 nodes could be used effectively in a purely data parallel sonar spectral detection system, primarily because the data set sizes are fairly small and cannot be increased.

The point is that, depending on the sizes of the machine and the data sets, the same source program can benefit from different styles of parallelism: data parallelism, task parallelism, or some combination of the two. In this paper, we discuss a practical approach to accomplish this. We have implemented a parallelizing Fortran

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. This material is based upon work supported under a National Foundation Graduate Research Fellowship.

compiler for the iWarp multicomputer [2, 3]. The key idea is that a *single source program* can be translated to fit a particular machine or data set size. The style of the generated code is based on compiler analysis and user hints.

We have used the compiler to explore the tradeoffs between task and data parallelism for some model programs operating on different size data sets. In particular, we compared a purely data parallel implementation and a pipelined data parallel implementation for two example programs, and observed that the best performance was obtained with different implementations for different data sizes.

Section 2 describes the programming model. Section 3 describes the compilation approach. Section 4 discusses the communication models used for generating data parallel and task parallel programs. Section 5 presents experimental results that compares the performance of programs compiled with different styles of parallelism. Section 6 discusses related work.

2 Programming model

The language we chose is Fortran 77 augmented with Fortran 90 array syntax and data layout statements based on Fortran D [1] and the emerging High Performance Fortran (HPF) standard [8]. The choice of Fortran was based on convenience and user acceptance. We have taken a source-to-source compilation approach; the output of the compiler is a Fortran 77 program (with calls to a communication library) for all the nodes in the system.

2.1 Data parallel constructs

Data parallelism is expressed using data layout statements, array syntax, and a simple parallel loop. For example:

```
C$      template t(n)
C$      align A(i,j) with t(i)
C$      align B(i,j) with t(j)
C$      distribute t(cyclic)

        do i=1,n
           A(i,:) = A(i,:) + B(:,i)
        enddo
```

This example uses `template`, `align`, and `distribute` directives to distribute the rows of array `A` and the columns of array `B` cyclically across the node array. In the example above, the *ith* loop iteration uses an array statement to add the *jth* column of `B` to the *ith* row of `A`. Moreover, each loop iteration is independent and can run in parallel with the other loop iterations.

2.2 Task parallel constructs

Task parallelism is expressed as special code regions called *parallel sections*. The body of a parallel section is limited to subroutines and loops. For ease of implementation, we require that these loops have constant bounds. The subroutines may contain data parallel operations. Each of these subroutine calls is followed by `input` and `output` directives, which identify the input and output parameters of the subroutine. The arguments in the `input` and `output` lists can be scalars, array slices with constant bounds, or whole arrays.

The body of a parallel section contains a task for each invocation of a subroutine call. We use subroutines to identify tasks because of programming and implementation convenience. A task is a single thread of control with well defined side effects. Tasks communicate with other tasks only through their input and output parameters. The communication is completely determined by the way the data would flow if the program was executed sequentially. A task waits for its inputs, executes, produces its outputs, and then terminates. Finally there is an implicit barrier synchronization at the end of a parallel section for data consistency. The following example illustrates task parallel programming:

```
C$      begin parallel
        do i = 1,10
           call src(A,B)
C$         output: A,B
           call p1(A)
C$         input: A
C$         output: A
           call p2(B)
C$         input: B
C$         output: B
           call sink(A,B)
C$         input: A,B
        enddo
C$      end parallel
```

This example consists of forty tasks, with ten tasks corresponding to each lexical subroutine call in the parallel section. During each iteration, a `src` task distributes array `A` to a task `p1` and array `B` to a task `p2`, which operate on their inputs and send their outputs to a `sink` task. The data dependences for one loop iteration are shown in Figure 1. Note that within each loop iteration, tasks `p1` and `p2` can execute in parallel, and that the loop iterations can be pipelined.

There are several important points to make about this task model. First, while the simple notion of a parallel section limits the class of parallel programs that can be expressed (e.g., communication between tasks is prohibited except on entry and exit), it allows for

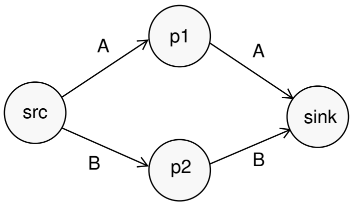

Figure 1: Data dependences for one loop iteration.

efficient implementation with relatively little analysis. Second, if the directives are correct and complete, then the parallel program has sequential semantics. Third, the compiler has the freedom to choose between several possible mappings of task parallelism onto a machine, and can even choose to ignore the directives and compile the program in a purely data parallel style.

3 Compilation approach

Figure 2 illustrates the main compilation steps. In this example, we start with a single source program consisting of two data parallel subroutines and a main program that calls them as tasks. The program is based on the 2D FFT model program discussed later. The compilation proceeds as independent data parallel and task parallel phases. The data parallel phase analyzes the array statements and parallel loops, and generates data parallel code for each subroutine called in the body of the parallel section.

The task parallel phase analyzes directives inside the parallel section and generates a layer of the hierarchical *uniform task graph (UTG)* that captures the dependence relationships between tasks. In this example, the top layer of the UTG consists of six tasks, two per loop iteration. The compiler then partitions the UTG into *modules*, forming a *module graph* where each node is a module, and each arc represents possible communication between nodes. A module is a unit for placement on the processor nodes and is compiled with the native Fortran 77 compiler. Finally, the modules are mapped to the physical machine, with each module assigned to a disjoint collection of nodes.

3.1 Compiling array statements

The core data parallel construct is the array assignment statement:

$$\mathtt{A}(f_1 : l_1 : s_1) = \mathcal{F}(\mathtt{B}(f_2 : l_2 : s_2))$$

where $\mathcal{F}$ is some element-wise intrinsic operator, and arrays `A` and `B` are assigned block-cyclic distributions using `template`, `align`, and `distribute` statements.

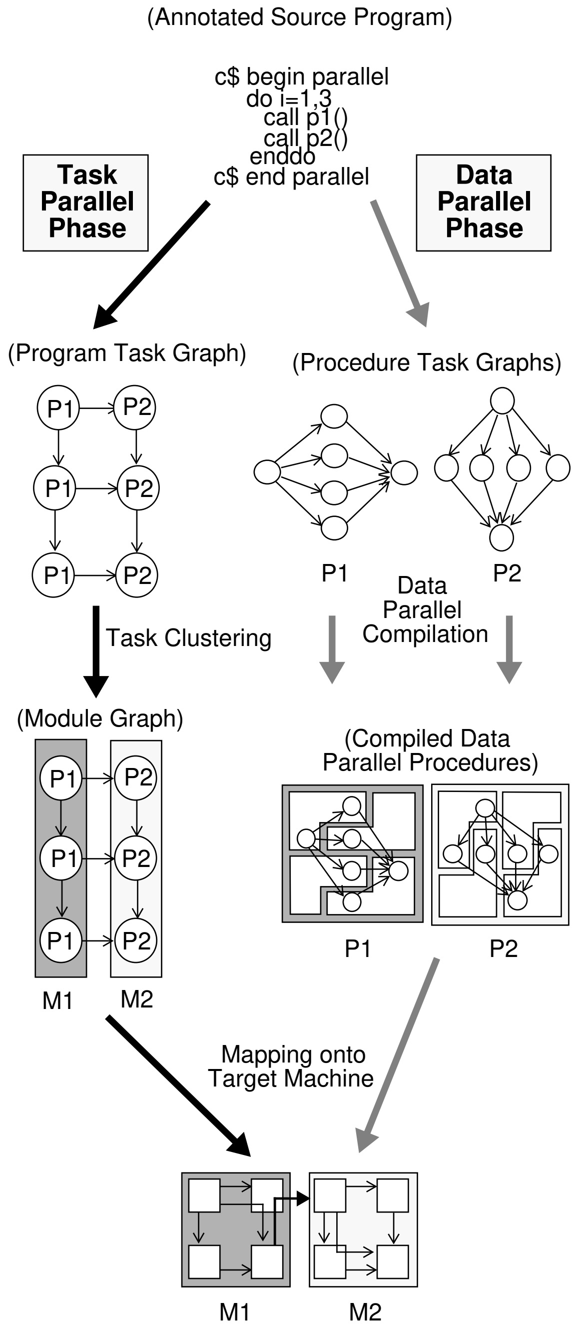

Figure 2: Compilation approach

Note that block distributions and cyclic distributions are both special cases of block-cyclic distributions.

The compiler determines the necessary communication for such a statement using the *owner-computes rule*, which states that the computation is distributed across the processing elements according to where the results are to be stored. In the best case, results and operands are stored on the same processor, and no communication is necessary. The worst case, however, is an all-to-all personalized communication, where every node must send data to every other node and must correspondingly receive data from every other node. Such communication is represented as edges in the UTG.

In the general case, it can be quite expensive to compute which array elements to send [17]: Consider the above assignment statement. When $f_2 = f_1$ and $s_2 = s_1$, the sending node s must send to the destination node d the elements of `B` determined by this index set:

$$Own_s(\mathtt{B}) \cap Own_d(\mathtt{A}) \cap (f_1 : l_1 : s_1)$$

where $Own_s(\mathtt{B})$ is the set of indices of elements of `B` owned by node s, and $Own_d(\mathtt{A})$ is the set of indices of those elements of `A` owned by node d. Node s should send $\mathtt{B}[i]$ to node d if and only if node s owns $\mathtt{B}[i]$, node d owns $\mathtt{A}[i]$, and i is one of the indices specified in the array slice $(f_1 : l_1 : s_1)$.

Consider for the assignment statement `A = B` (i.e., $f_1 = f_2 = s_1 = s_2 = 1$). Assume that both arrays have a block-cyclic distribution over two nodes, where `A` has a block size of 3 and `B` has a block size of 5. Figure 3 depicts the data transfers required for this statement. The shaded boxes in the middle row represent the elements of `B` owned by the sending node s, and the shaded boxes in the top and bottom rows represent the elements of `A` owned by the two receiving nodes d_1 and d_2, respectively. The arrows, which represent the intersections of the corresponding ownership sets, show the elements with common indices; these elements must be communicated. Note the irregularity, even though the block-cyclic distribution patterns are regular and easy to represent, as are the array slices in the assignment statement. This kind of communication is not uncommon, as such assignment statements are used to redistribute arrays. The situation becomes even more complex when we introduce non-unit strides s_1 and s_2 into the assignment statement.

Figure 3 illustrated the sender's task of determining which elements of an array to send. It is equally expensive for a node to compute the set of elements to receive. Although the algorithm to determine the elements to send is nearly identical to the algorithm to determine the elements to receive, in the general case, the respective algorithm must be executed on both the sending and receiving nodes. The similarity between the algorithms motivates the optimizations presented in Section 4.1.

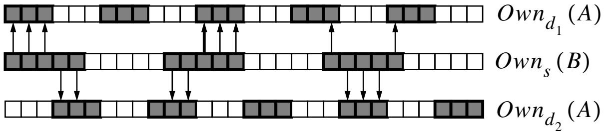

Figure 3: Communication for `A = B`, for block-cyclic distribution of `A` (block size 3) and `B` (block size 5)

3.2 Compiling parallel tasks

For task parallelism, the compiler has to address another set of issues. Most importantly, precise data dependence and communication relationships between tasks must be collected at compile time. Since the variables that a subroutine can access and modify are explicitly described in the `input` and `output` directives (including arrays and array sections), and since there are no conditional statements inside a parallel section, a data flow framework on array sections can statically determine the dependence and precise data movement between tasks.

The second major issue is clustering of tasks into modules. Task clustering determines the amount of computation in each module. In our current implementation, all tasks corresponding to a lexical procedure call constitute a module. This is a simple model that is based on the notion of treating a subroutine as a unit of parallel computation, and fits many applications.

Finally, the modules must be mapped onto the physical machine. This includes allocation of processor nodes and placement of modules on the node array. Allocation of nodes to modules determines the load balancing and is a critical factor for efficient execution. In the current implementation this is controlled explicitly by the programmer through directives. However, the compiler does provide a framework where a programmer can experiment with different processor allocations, simply by changing directives in the program.

4 Communication

An iWarp node is physically connected to 4 other nodes by 8 unidirectional *communication buses*. The iWarp communication system provides *logical channels* between adjacent nodes [3]. Data sent over different logical channels is kept separate. A *pathway* is a sequence of logical channels; pathways are also directly supported by the communication system and provide for "long-distance" connections between arbitrary pairs of nodes in the iWarp system. Like logical channels, pathways can be created and destroyed at runtime, with low overhead. Node programs can access pathways directly, one word at a time, without intervention from the runtime system.

Pathways are the fundamental building blocks for implementing a variety of communication styles on iWarp. An important style for the compiler is the *permanent network*, which is a collection of pathways created at the beginning of program execution and left in place for the duration of the program.

Pathways can also be used to implement a variety of message passing styles. The basic idea is to create a *transient pathway* that exists only for some duration during the execution of a program. One usage model is to transfer only a single message over a transient pathway . Each time there is a message to send from node n_i to node n_j, the program creates a pathway between the two nodes, transfers the data over the pathway, and then destroys the pathway. Different message passing styles can be constructed in this way by varying the behavior at the endpoints of the individual pathways.

In our compiler, permanent networks are used to transfer data between parallel tasks and to perform special-purpose communication primitives such as barrier synchronization and broadcasting. Message passing based on transient pathways is used for all other communications. The compiler partitions the communication resources on each node into two sets; one set is used exclusively for permanent networks, and the other set is used exclusively for message passing. The important points here are that the same pathway mechanism is used for all communication styles, and that the pathways can be manipulated directly by the user–level program, without intervention from the runtime system.

4.1 Data parallel communication

As discussed in Section 3.1, the compiler manages the movement of data from one node to another. Consider an element `B[i]` of an array `B` that is required for a computation on some node N. If `B[i]` is not stored on N, it must be moved to `B'[j]` for some array `B'` such that `B'[j]` is local to N. In general, the exact communication patterns for a data parallel program are not known at compile time, and therefore, some form of general-purpose message passing is required.

Several different message passing styles have been identified: The message passing system can buffer `B[i]` in temporary storage on the destination node, and then `B[i]` is moved from that buffer to the location `B'[j]`, where the application program expects it. This is an example of the station-to-station communication style [2]. However, such copying is wasteful; another style is to allow the communication system to place `B[i]` directly into `B'[j]` (door-to-door delivery [2]).

The sender computes the addresses of the elements `B[i]` to be transmitted, but note that in general $\mathtt{i} \neq \mathtt{j}$. So for both communication styles, it is necessary to compute the index `j` for `B'[j]` This computation of `j` shares common subexpressions with the computation of `i`. To exploit these common computations, we developed a new communication style called *deposit model message passing*. The central idea behind deposit model message passing is that the sending node computes the destination addresses of the data on the receiving node, and passes these addresses along with the data. Rather than computing the destination addresses explicitly, the receiving node merely reads the addresses and deposits the received data into the appropriate memory locations. The key point is that the sending node computes the destination addresses, which eliminates the address computation on the receiver while imposing only a small additional burden on the sender.

Deposit model message passing incurs an additional cost. Since addresses must be passed along with the data, some part of the communication bandwidth is wasted. This overhead can range from small, when regular sections of the array can be sent in large blocks, to large, in the case of address-data pairs, when an explicit destination address is sent for each data item. This reduces the effective bandwidth of the communication system by at most a factor of two. However, in our experience, this is more than compensated for by the reduced workload on the receiving nodes.

Deposit model message passing is a general-purpose communication style that is not specific to iWarp. It can be used effectively by any parallelizing compiler for any private memory computer system. However, on iWarp, the direct word-level access to the pathways allows for another important optimization. The receiving node scatters the data to memory *directly*, as each data item arrives over the communication bus. The scattering occurs at the full 40 Mbytes/second bandwidth of the communication buses.

4.2 Task parallel communication

A program consists of a set of modules, each executing on a subset of available nodes. Each module consists of a set of tasks that potentially communicate, but only on entry and exit. In our model, a module is independent of other modules, and in the current implementation, modules are placed on disjoint sets of nodes on an iWarp machine. The interface between two modules is the message format used to transfer the input and output parameters.

Communication between modules conceptually consists of collecting the data in the sending module, send-

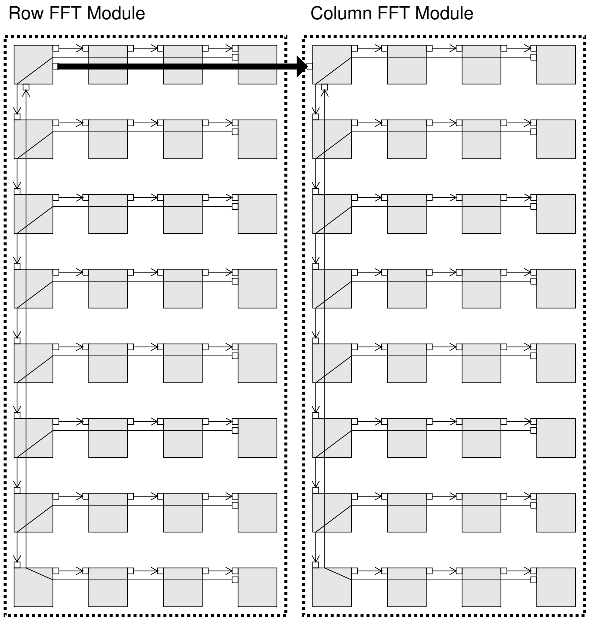

Figure 4: Networks for task parallel communication.

ing the data to the receiving module, and distributing the data in the receiving module. A direct implementation would require the existence of nodes that have enough memory to hold all the data to be transferred. Also, at least 6 memory operations (a read and a write each for gathering, transferring, and distributing) would be required to transfer each word of data.

In the actual implementation, we create a permanent *snake* network in each module, as shown in Figure 4. We forward data on the snake network in a synchronized manner, such that the data reaches a designated *master* node in a canonical order. The master node, which is connected to the master node of the receiving module by a permanent inter-module pathway (shown as a thick arrow in Figure 4), simply forwards the data directly from the snake network to the intermodule pathway, and ultimately to the master node of the receiving module, *without ever placing the data in memory*. The data is distributed in a synchronous fashion similar to the way it is collected. This implementation requires only one memory read and write for every data element, and the data transfer is pipelined. Data can be transferred at the peak bandwidth of a single logical channel.

The bandwidth of one communication bus is an important limiting factor for this approach, and can limit the overall performance of an application. This phenomena was observed in the experiments that we discuss in the next section. Clearly, the performance can be further improved by transferring data in parallel between individual nodes in the modules, using, for instance, message passing to send individual data objects.

However, in our implementation model, a module is an independent process with a standard communication interface. The processing elements inside a module do not have any knowledge of the physical location or data distribution of other modules. This property is extremely important in situations where an application consists of several modules executing on different machines on a network, or when dynamic load balancing can change the computation and data distribution associated with a module. Thus, we have chosen a model that is applicable to a wide variety of computation environments, and is not tailored for our current implementation.

5 Experimental results

Depending on the size of the input sets and the machine, compiling with one style of parallelism may allow the source program to run faster than with another style. To illustrate the benefits that can be obtained from choosing between different styles of parallelism at compile time, we investigated a small but important example: the 2-dimensional Fast Fourier Transform (2D FFT). We first examine the implementation of a 2D FFT program, and then a program in which a 2D FFT is followed by a stage that analyzes the results and generates histograms.

5.1 Example: 2D FFT

The 2D FFT example inputs a sequence of m $n \times n$ complex arrays, and performs a 2D FFT on each array in sequence. This models the situation in real-time signal or image processing applications in which a stream of input comes from a collection of sensors. Each 2D FFT computation consists of two passes over the array. The first pass performs an independent 1D FFT on each of the n rows of the array. The second pass performs an independent 1D FFT on each of the n columns of the array. The output is an $n \times n$ array.

We chose the 2D FFT because of its intrinsic importance, because it is similar in its structure to many larger signal processing applications such as synthetic aperture radar imaging and sonar adaptive beam interpolation, and because it is a satisfying combination of the easy and the challenging: easy because the 1D FFT's on the rows (columns) are independent; challenging because it must scan both the rows *and* the columns of the input array, and therefore requires an efficient redistribution of data to achieve good performance. The Fortran program is of the form:

```
C$      begin parallel
        do i = 1,m
```

```
            call rffts(A)
C$          output:  A
            call cffts(A)
C$          input:  A
         enddo
C$       end parallel
```

The program, 2DFFT, consists of a parallel section, with calls to two subroutines inside a loop that iterates m times. Subroutine *rffts* is a data parallel procedure that produces a complex $n \times n$ array A, and then applies a 1D FFT to each row of the matrix. Subroutine *cffts* is a data parallel task that applies a 1D FFT to each column of array A, and then verifies the correctness of each element of the result matrix. The source of parallelism inside each task is a parallel loop statement.

To evaluate the impact of the input data size on the relative performance of data parallelism and task parallelism, we used the compiler to automatically map the same 2DFFT program onto a 64-node iWarp system in two different ways:

DP–2DFFT: *Purely data parallel.* The directives for task parallelism are ignored and the program is compiled as a single thread of control running on all 64 nodes. During each iteration of the outer loop, all nodes compute the row FFT's, transpose the data, and then compute the column FFT's. The parallelism is in the form of independent parallel loop iterations within each subroutine.

PT–2DFFT: *Pipelined tasks.* The same source program is compiled as a collection of pipelined data parallel tasks. Each task is compiled in exactly the same style as DP–2DFFT, but for 32 nodes rather than 64 nodes. The tasks that update the rows are clustered into one module and placed on 32 nodes. Similarly, the tasks that update the columns are clustered into another module and placed on the remaining 32 nodes. The parallelism is in the form of data parallel loops within each module, and pipelining between the two modules.

We do not consider a purely task parallel style, because for this example there are not enough tasks to keep the nodes busy.

We measured the performance of these two styles for different input sizes. Figure 5 compares the performance of the data parallel implementation with the performance of the pipelined tasks implementation. We observe that the pipelined tasks (PT–2DFFT) execute faster for the smaller data sizes, in this case $n < 256$. For $n \geq 256$, the purely data parallel style (DP–2DFFT) outperforms the pipelined data parallel version. The figure shows a crossover point near 200.

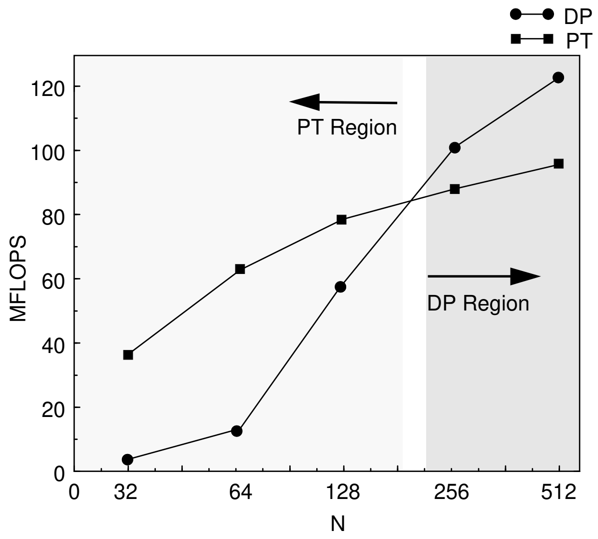

Figure 5: Performance of $N \times N$ 2DFFT on 64 nodes.

Figure 6 shows the measured performance as more processors are added to a fixed size 2D FFT problem ($n = 128$). We observe that the performance of both data parallel (DP–2DFFT) and pipelined tasks (PT–2DFFT) implementations is similar and improves approximately linearly as the number of processors is increased, up to 32 processors. However, when the number of processors is doubled to 64, the pipelined tasks version again shows the expected improvement, but the purely data parallel version shows a small improvement. This illustrates that, for some problem sizes, a fixed size 2D FFT problem can use a larger number of processors effectively when executing as pipelined tasks.

5.2 Example: FFT–HIST

Our second example consists of a program containing a 2D FFT computation, followed by a result analysis stage which generates histograms and performs some other local computations. The Fortran program is of the form:

```
C$       begin parallel
         do i = 1,m
            call fft(A)
C$          output:  A
            call hist(A)
C$          input:  A
         enddo
C$       end parallel
```

We again build two versions of the same program for a 64-node iWarp system using the compiler. In the purely data parallel version (DP–FFT-HIST), a single thread of control executes on all 64 nodes. During each iteration of the outer loop, all nodes first compute the row

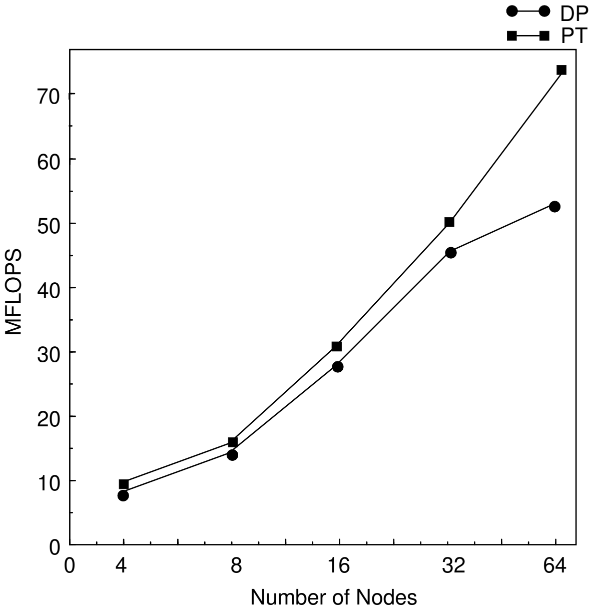

Figure 6: Performance of 2DFFT as a function of number of nodes.

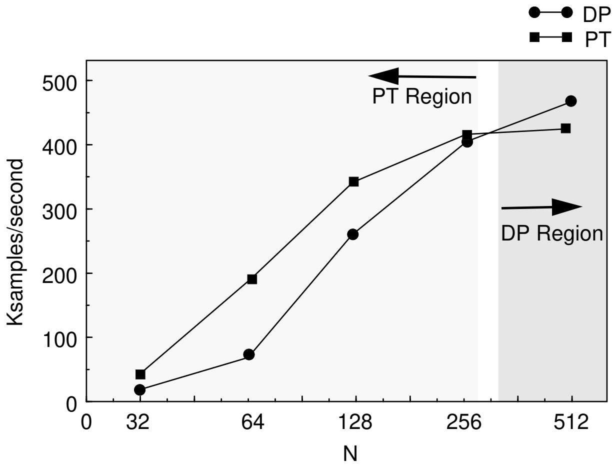

Figure 7: Performance of $N \times N$ FFT–HIST on 64 nodes.

FFT's, transpose the data, and compute the column FFT's. Next all nodes perform local analysis and collect local data for histograms. Finally histograms are generated using reductions over all processors. In the pipelined tasks version(PT–FFT-HIST), the program is compiled as two sets of tasks, each running on 32 nodes. Each task in the first set performs a 2D FFT computation, and each task in the second set generates histograms. Computation inside the tasks is data parallel.

It is important to note that there is a significant qualitative difference between this example and the previous one. The two stages in this example (i.e., a 2D FFT and a Histogram) each require significant amount of internal communication, while the stages in the previous example (i.e., 1D FFT's) required no internal communication.

The performance results obtained by executing the two versions of the program are shown in Figure 7. Note that since this computation is not dominated by floating point operations, our metric is the number of samples processed per second. We again observe that the results are qualitatively similar to those obtained for the previous example, The pipelined tasks version has better performance for data sizes up to around 300, and the data parallel version has better performance for larger data sizes.

5.3 Discussion of performance results

It is informative to analyze the results for the two examples. The first example consists of two separate 1D FFT stages. In the purely data parallel mode, each 1D FFT stage executes on all nodes in parallel, with no communication within each stage, but on transition between the two stages, a matrix transpose operation must be performed. Using pipelined tasks, the two stages are also data parallel, but in addition, the stages are pipelined. The transpose operation is merged with the transfer of data from one set of nodes to the other set of nodes, so there is no need for an explicit matrix transpose operation.

The main advantages of a task parallel approach in this example are:

- On each node, twice as many operations are performed between communication operations, since the same computation is mapped onto half as many nodes.
- For relatively small data sizes, transfer of data between disjoint sets of nodes can be implemented more efficiently than a matrix transpose.

For these reasons, the pipelined tasks version outperforms the purely data parallel version for problem sizes up to around 200. For larger problem sizes, the above factors are less important because the granularity of computation and communication increases, and the ratio of computation to communication ($O(\log n)$) increases noticeably. Most importantly, the fact that there is sequential communication between the two tasks along a channel with a fixed bandwidth (40Mbytes/sec) becomes a dominant factor, since all the data parallel communication is parallelized, even though it may have a higher overhead.

In the second example, there is significant global communication within each stage. Specifically, the 2DFFT task contains an all-to-all communication in the matrix transpose operation, and the Histogram task uses a set of reductions for generating histograms. While the number of communication steps is exactly the same in both implementations, all global communication in the pipelined tasks implementation is performed over half the number of cells. This reduces the communication overhead, particularly for small problem sizes where communication overhead tends to dominate. For this reason, pipelined tasks outperform purely data parallel implementations in this example for a larger range of data sizes.

We would like to point out that the smaller input sizes, for which the pipelined tasks version outperforms the purely data parallel version, are certainly of practical interest. For example, a sonar adaptive beam interpolation (ABI) application manipulates a sequence of $r \times c$ complex arrays. Like 2D FFT, ABI first manipulates the rows of the matrix, and then manipulates the columns. The parameter r is proportional to the number of sensors attached to a cable that is towed behind a ship, and c is the number of frequencies of interest. While c is typically large, on the order of 1000, r is typically small, on the order of 50. It is limited by cost and length of the cable, and simply cannot be increased to fit the number of nodes.

Often the only way to determine whether a purely data parallel implementation or a pipelined tasks implementation is more suitable for a particular computing situation is by experimentation. Therefore, we argue that it is extremely useful to have a system to experiment with the tradeoffs involved. This is especially important for problems that are dominated by global communication because communication characteristics during execution can change significantly when task parallelism is introduced in addition to data parallelism.

6 Related work

Compiling for private memory parallel computers has been an active area of research in the recent past and several different approaches have been discussed in the literature. We compare our approach to specific research projects using other approaches, and discuss their influence on this project.

Several research and production compilers have been built for compiling a program in data parallel fashion to a SPMD parallel program. These include the Fortran D compiler [10], SuperB [20], the Vienna Fortran compiler [5], and the Thinking Machines Fortran compiler [1]. The general approach is to take a sequential program in a conventional language (Fortran for the above three), possibly with directives to guide data layout and parallelism. The compiler parallelizes the program, typically using the owner computes rule to distribute the computation among nodes. We have also taken this approach for exploiting data parallelism. The data layout statements are similar to those used in Fortran D, Vienna Fortran, and HPF. However, we also exploit task parallelism, which is not addressed in these projects.

Assign [14] is a programming tool that is used to develop task parallel programs. The user specifies the parallel tasks and the communication between them, and the tool maps the tasks to physical nodes. Jade [12] extends this approach by allowing the programmer to augment the code with abstract data usage information instead of specifying communication explicitly. Our research has been motivated by the Assign project, and some aspects of our design are similar to Jade, but there are several important differences. We use an existing language and do not introduce any new language constructs. All auxiliary information is in the form of compiler directives, and therefore the base program is an ordinary sequential program. Furthermore, we also address data parallelism within the same language and compiler model.

One approach to exploiting data and task parallelism is to use a coordination language for expressing task parallelism. SCHEDULE [6] allows the user to express a set of tasks and their dependence relationships, and the system implements the task parallelism. PVM [18] provides user primitives that can be used to build heterogeneous parallel applications from components, which can potentially be data parallel tasks. Linda [4] provides a programming paradigm which can be used for expressing task parallelism, while the data parallelism may be expressed in a different data parallel language. Our approach differs from others discussed above in that we use the same framework for both task and data parallelism, and build on existing technology for parallelization of data parallel programs. We provide support for both kinds of parallelism inside Fortran programs, and a single compiler manages all parallelism. This is important for two reasons. First, it simplifies the task of writing a parallel program, since the user must deal with only one programming paradigm. Second, the compiler has the opportunity to manage the tradeoffs between task and data parallelism, as illustrated by our example programs.

7 Concluding remarks

Many real-world applications must exploit both task and data parallelism to achieve good performance. The size of the input data set is an important parameter that influences which style of parallelism is more appropriate for mapping a given computation onto a parallel system. We have developed a single framework that makes programming with task and data parallelism convenient: the user can develop programs in a single environment, and the compiler can make tradeoffs between task and data parallelism. Our measured results are preliminary, but it is becoming clear that it is important to exploit both task and data parallelism.

8 Acknowledgements

We would like to thank Susan Hinrichs for developing the iWarp's PCS communication tool chain, Thomas Stricker, Thomas Warfel, and Michael Hemy for their advice on message passing, Bwolen Yang for assistance with the FFT programs and iWarp measurements, Jim Wheeler at GE and Tom Stover at NAWC for teaching us about sonar signal processing, Jon Webb for advice on parallel loops, and Keith Bromley at NOSC for encouraging us to pursue a unified framework for task and data parallelism.

References

[1] Albert, E., Knobe, K., Lukas, J., and Steele, G. Compiling Fortran 8x array features for the Connection Machine computer system. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems* (New Haven, CT, July 1988), pp. 42–56.

[2] Borkar, S., Cohn, R., Cox, G., Gleason, S., Gross, T., Kung, H. T., Lam, M., Moore, B., Peterson, C., Pieper, J., Rankin, L., Tseng, P. S., Sutton, J., Urbanski, J., and Webb, J. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing '88* (Nov. 1988), pp. 330–339.

[3] Borkar, S., Cohn, R., Cox, G., Gross, T., Kung, H. T., Lam, M., Moore, M. L. B., Moore, W., Peterson, C., Susman, J., Sutton, J., Urbanski, J., and Webb, J. Supporting systolic and memory communication in iWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, WA, May 1990), pp. 70–81.

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

[5] Chapman, B., Mehrotra, P., and Zima, H. Programming in Vienna Fortran. *Scientific Programming 1*, 1 (Aug. 1992), 31–50.

[6] Dongarra, J., and Sorensen, D. A portable environment for developing parallel Fortran programs. *Parallel Computing 5* (1987), 175–186.

[7] Fox, G. The architecture of problems and portable parallel software systems. Tech. Rep. CRPC-TR91172, Northeast Parallel Architectures Center, 1991.

[8] High Performance Fortran Forum. *High Performance Fortran Language Specification*, Jan. 1993. Version 1.0 DRAFT.

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

[10] Hiranandani, S., Kennedy, K., and Tseng, C. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91* (Albuquerque, NM, November 1991), pp. 86–100.

[11] Koelbel, C., Mehrotra, P., and Rosendale, J. V. Semi-automatic domain decomposition in BLAZE. In *Proceedings of the 1987 International Conference on Parallel Processing* (August 1987), S. K. Sahni, Ed., pp. 521–524.

[12] Lam, M., and Rinard, M. 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), pp. 94–105.

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

[14] O'Hallaron, D. The Assign parallel program generator. In *Proceedings of the 6th Distributed Memory Computing Conference* (Portland, OR, Apr. 1991), pp. 178–185.

[15] Printz, H. *Automatic Mapping of Large Signal Processing Systems to a Parallel Machine*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1991. Also available as report CMU-CS-91-101.

[16] Printz, H., Kung, H. T., Mummert, T., and Scherer, P. Automatic mapping of large signal processing systems to a parallel machine. In *Proceedings of SPIE Symposium, Real-Time Signal Processing XI* (San Diego, CA, Aug. 1989), Society of Photo-Optical Instrumentation Engineers, pp. 2–16.

[17] Stichnoth, J. Efficient compilation of array statements for private memory multicomputers. Tech. Rep. CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, Feb. 1993.

[18] Sunderam, V. S. PVM : A framework for parallel distributed computing. *Concurrency: Practice and Experience 2*, 4 (December 1990), 315–339.

[19] Tseng, P. S. *A Parallelizing Compiler For Distributed Memory Parallel Computers.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1989. Also available as CMU Tech Report CMU-CS-89-148.

[20] Zima, H., Bast, H.-J., and Gerndt, M. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing 6* (1988), 1–18.