Automatic Mapping of Task and Data Parallel Programs for Efficient Execution on Multicomputers

Jaspal Subhlok

November 1993
CMU-CS-93-212

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

For a wide variety of applications, both task and data parallelism must be exploited to achieve the best possible performance on a multicomputer. Recent research has underlined the importance of exploiting task and data parallelism in a single compiler framework, and such a compiler can map a single source program in many different ways onto a parallel machine. There are several complex tradeoffs between task and data parallelism, depending on the characteristics of the program to be executed and the performance parameters of the target parallel machine. This makes it very difficult for a programmer to select a good mapping for a task and data parallel program. In this paper we isolate and examine specific characteristics of executing programs that determine the performance for different mappings on a parallel machine, and present an automatic system to obtain good mappings. The process consists of two steps: First, an instrumented input program is executed a fixed number of times with different mappings, to build an execution model of the program. Next, the model is analyzed to obtain a good final mapping of the program onto the processors of the parallel machine. The current implementation is static, feedback driven, although the approach can be extended to a dynamic system. We demonstrate the system with an example program that is a model for many applications in the domains of signal processing and image processing.

This research was sponsored by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

Keywords: Parallel programming, task parallelism, parallelizing compilers, load balancing, automatic mapping, processor allocation

1 Introduction

Many applications can be naturally expressed as collections of coarse grain tasks, possibly with data parallelism inside them. The Fx compiler at Carnegie Mellon is designed to allow the programmer to express and exploit task and data parallelism, and we have demonstrated the power and value of this approach [12]. In this paper, we address the problem of mapping a task and data parallel program to achieve the best performance.

There are several reasons to support task parallelism, in addition to data parallelism, in a parallelizing compiler. For many applications, particularly in the areas of digital signal processing and image processing, the problem sizes are relatively small and fixed, and not enough data parallelism is available to effectively use the large number of processors of a massively parallel machine. Even when adequate data parallelism is available, a data parallel mapping may not be the most efficient way to execute a program. A purely data parallel approach forces all computations to be performed on a fixed set of processors, whose number typically varies from tens to thousands. However, it is the amount and nature of parallelism and communication, that determine how well a computation scales as the number of processors is increased. A complete application often consists of a series of computations with different computation and communication requirements. For best performance, it may be necessary to assign different sets of processors to different computations, and execute several different computations simultaneously.

Using task and data parallelism together, it is possible to map an application in a variety of ways onto a parallel machine. Consider an application with three coarse grain, pipelined, data parallel computation stages, with each execution of a computation stage corresponding to a task. A set of mappings for such an application is shown in Figure 1. Figure 1(a) shows a pure data parallel mapping, where all processors participate in all computation stages. Figure 1(b) shows a pure task parallel mapping, where a subset of processors is dedicated to each computation stage. It may be possible to have multiple copies of the data parallel mapping executing on different sets of processors, as shown in Figure 1(c). Finally, a mix of task and data parallelism, with replication is shown in Figure 1(d).

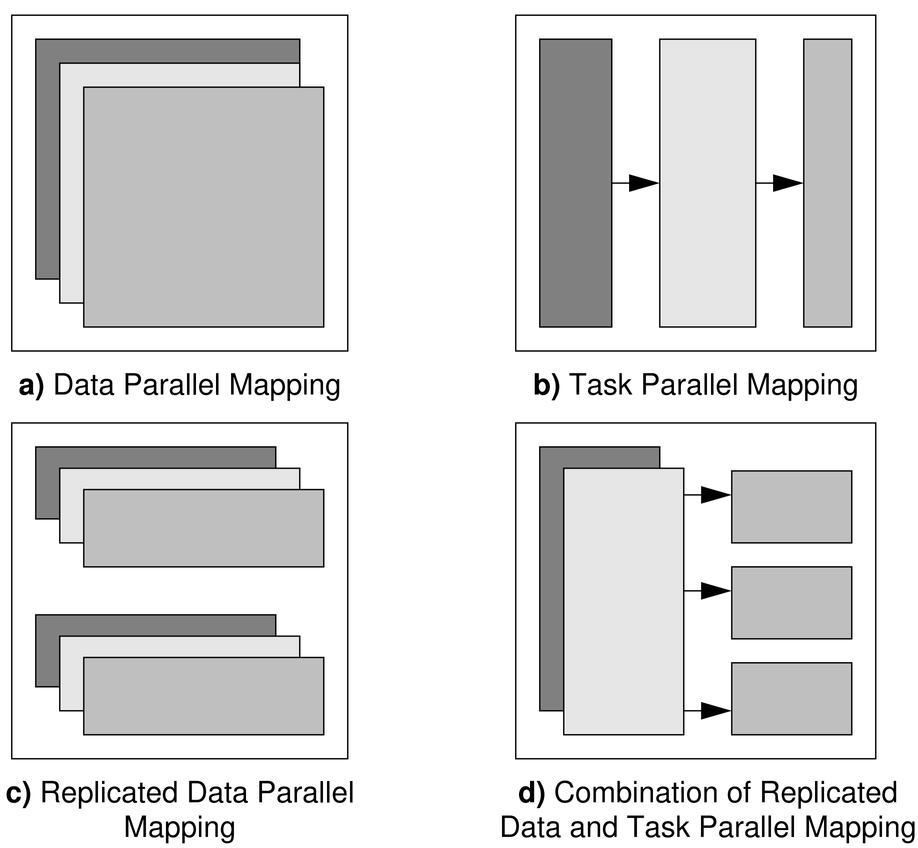

Figure 1: Combinations of Data and Task parallel mappings

The fundamental question that we are addressing in this paper is: "Which of the many mappings that are feasible when data and task parallelism are used together is optimal, and how can this be determined automatically ?"

A key determinant of performance is *communication locality* within each data parallel stage, but it has to be traded off against processor usage, memory requirements, and inter-task communication cost. Communication locality can be improved by executing each task on a small number of processors, but this makes it difficult to load balance and keep all processors busy, implies a higher memory requirement per processor, and possibly a higher inter-task communication cost, due to a finer task granularity.

Our solution is to build an execution model of the application using timing information from trial executions, and analyze it to predict the most efficient mapping. Our current implementation is static and feedback driven, and is applicable to programs for which sample data sets can capture the general execution behavior. In a dynamic system, the scope is extended to other programs where recent execution history is a good predictor of future execution behavior.

2 Overview

2.1 Programming and compiling task parallelism

The Fx compiler supports task and data parallelism [12]. The base language is Fortran 77 augmented with Fortran 90 array syntax, and data layout statements based on Fortran D and High Performance Fortran. Data parallelism is expressed using array syntax and parallel loops. The main concepts in the Fx compiler are language independent, and Fortran was chosen for convenience and user acceptance. The current target machine is an iWarp processor array [2].

Task parallelism is expressed in special code regions called *parallel sections*. The body of a parallel section is limited to calls to subroutines called *task-subroutines*, with each execution instance representing a parallel task, and loops to represent multiple instances. Each task-subroutine call is followed by `input` and `output` directives, which define the interface of the task subroutine to the calling routine, that is, they list the variables in the calling routine that are accessed and modified in the called task-subroutine. The entries in the input and output lists can be scalars, array slices, or whole arrays. The task-subroutines may have data parallelism inside them. A parallel section of an example program is shown in Figure 2.

```
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
```

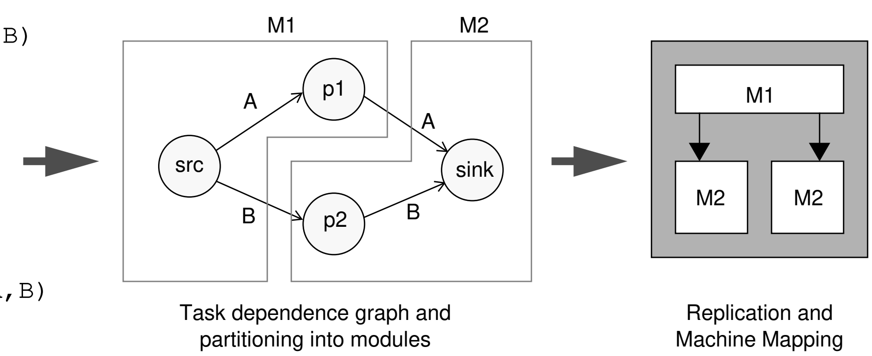

Figure 2: Compilation of task parallelism

The compiler can map and schedule instances of task-subroutines in any way, as long as sequential execution results are guaranteed[1]. During compilation, first the input and output directives are analyzed and a task level data dependence and communication graph is built. Next, the task-subroutines are grouped into *modules*. All task-subroutines in the same module are mapped to the same set of processors. The modules may be *replicated* to generate multiple modules instances, with each instance receiving and sending data sets in round robin fashion. Finally a subset of the processor array is assigned to each module instance.

[1] Assuming that input and output parameters are specified correctly

Figure 2 shows the application of these steps on a small example program. The task level dependence graph is built from the program and then partitioned into modules M1 and M2. Module M2 is replicated to two instances, and the program is mapped onto the machine.

2.2 Automatic Mapping

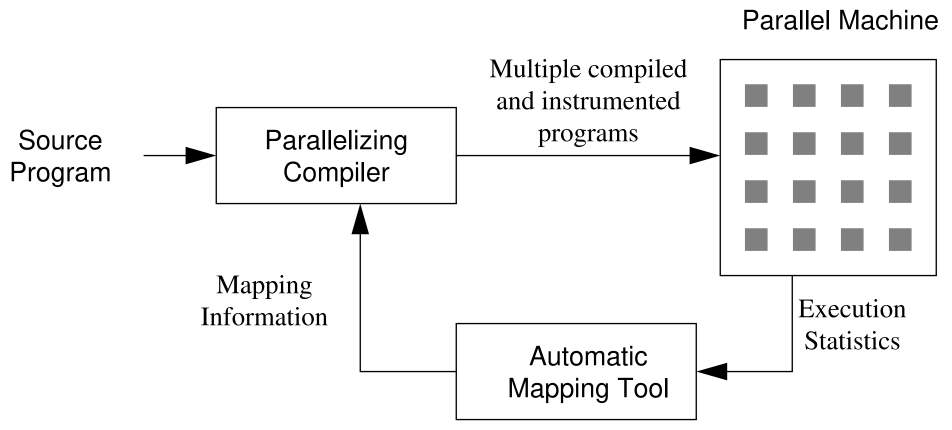

Figure 3: Automatic Mapping System

The subject of this paper is automation of the mapping process discussed in the previous subsection, which is otherwise driven by user assertions. The structure of the automatic mapping tool and process is shown in Figure 3. The programmer provides a source program with data parallel and task parallel constructs, as well as a sample data set. The mapping tool and the parallelizing compiler generate multiple instrumented versions of the program, which are then executed on the parallel machine, generating runtime statistics. The mapping tool uses these statistics to build an internal execution model of the program, and analyzes it to generate the final mapping information. This information is provided to the parallelizing compiler to generate a final parallel program for execution.

3 Execution model for task and data parallel programs

An Fx parallel program consists of a set of data parallel task-subroutines, that are related by data dependences between them. Every active task-subroutine is in one of the following states:

1. Waiting for a sender to be ready to send input.
2. Receiving input.
3. Executing.
4. Waiting for a receiver to be ready to receive output.
5. Sending output.

By measuring the time it spends in each of these states, for different mappings, we can determine the fundamental execution properties of a task-subroutine, which can in turn be used to predict the execution behavior for other mappings. In this section, we present an execution model for task-subroutines, which consists of parameterized equations for execution time, memory requirement , and inter-task communication time.

3.1 Execution time

The execution time of a task-subroutine on a set of processors is mainly determined by the total amount of computation, the amount of available parallelism, and the communication cost incurred in exploiting the parallelism. Thus, the execution time τ_x of task-subroutine T executing on P processors can be expressed as:

$$\tau_x(P) = C_x^0 + (C_x^1/P) + (C_x^2 * P)$$

The values of the C_x^i parameter describes the execution time characteristics of a specific task-subroutine. The constant term C_x^0 reflects the time spent on sequential computation in the program, as well as the fixed part of the communication cost. The second term represents the time spent in parallel computation, which decreases linearly with the number of processors. The last term represents the part of the communication cost that varies with the number of processors, as the parallelism in communication increases, but the granularity of communication becomes finer.

3.2 Inter-task communication time

The transfer of data between a pair of task-subroutines is of two different forms. If the two task-subroutines are mapped on the same set of processors, the data transfer is a potential local redistribution of data. If the task-subroutines are mapped on different sets of processors, there is movement of data from one processor subarray to another processor subarray. In both cases, the cost is dependent on the volume of the data to be transferred, and the number of processors involved. Increasing the number of processors implies a potentially higher degree of parallelism in communication, but may also increase the associated overhead.

We model the data transfer cost between task-subroutines executing on the same set of P processors (data parallel style) as:

$$\tau_{dp}(P) = C_{dp}^0 + (C_{dp}^1 * P)$$

And for a pair of task subroutines executing on P_1 and P_2 processors respectively (task parallel style) as:

$$\tau_{tp}(P_1, P_2) = C_{tp}^0 + (C_{tp}^1 * P_1) + (C_{tp}^2 * P_2)$$

3.3 Memory requirement

The memory requirement of a task-subroutine is an important parameter. In a multicomputer that has nodes with a fixed amount of uniform storage, the memory requirement determines the set of feasible mappings. In the presence of a memory hierarchy, the memory requirement plays an important role in determining overall performance.

The memory requirement is closely related to the way a parallelizing compiler allocates and manages memory. We outline the main memory requirements of a parallel program, and state how they are managed by our compilation system:

1. Global and system variables: Allocated for the duration of execution.

2. Local variables: Allocated on the stack as execution proceeds.

3. Compiler buffers and variables: Allocated dynamically on the heap as execution proceeds.

Global and system variables are allocated identically for all modules and processors. Local variables are allocated at run time on the stack as the task-subroutines are executed. Sufficient memory for the communication and other buffers is allocated on entry to a subroutine, and deallocated on exit. If an array argument is to be redistributed as a result of a a subroutine call, a compiler variable related to the size of the argument is allocated for the duration of the execution of the subroutine. If the argument is to be transferred from a task-subroutine that belongs to another module, such a compiler variable is not needed, since the argument is placed in the appropriate distribution on transfer between modules.

Consider the memory requirement of a processor that is executing a module M with task-subroutines $T_1, T_2, \ldots T_n$. If the local memory requirement of task-subroutine T_i is $\mu_{lm}^{T_i}$ and the global memory requirement for the program is μ_{gm}, then the total memory requirement μ_{total} for module M is:

$$\mu_{total} = \mu_{gm} + max_{i=1,n}(\mu_{lm}^{T_i})$$

max is taken since local memory for subroutines is allocated on stack. If a module is executing on a set of P processors, memory required per processor to hold global variables is:

$$\mu_{gm}(P) = C^0_{gm} + (C^1_{gm}/P)$$

where C^0_{gm} and C^1_{gm} represent the memory requirement due to replicated and distributed variables respectively.

The memory requirement of individual task-subroutines is dependent on arguments that are redistributed on subroutine entry. If the task subroutine that sends the argument is part of the same module, then additional memory is allocated for redistribution. If a task-subroutine has k arguments that are potentially redistributed on entry, then the per processor memory requirement for holding local variables and parameters can be modeled as:

$$\mu_{lm}(P) = C^0_{lm} + (C^1_{lm}/P) + \sum_{i=1}^{k}((b_i * C^i_{arg})/P)$$

where C^0_m and C^1_m/P represent the memory requirements of local replicated and distributed variables respectively, C^i_{arg} is the size of the ith redistributable argument and b_i is 1 if the task-subroutine from which the argument is sent is in the same module as T, and 0 otherwise.

4 Deriving the execution model parameters

We restate the parameterized equations for task execution time and inter-task communication time, from the previous section:

$$\begin{aligned}
\tau_x(P) &= C^0_x + (C^1_x/P) + (C^2_x * P) && (1)\\
\tau_{dp}(P) &= C^0_{dp} + (C^1_{dp} * P) && (2)\\
\tau_{tp}(P_1, P_2) &= C^0_{tp} + (C^1_{tp} * P_1) + (C^2_{tp} * P_2) && (3)
\end{aligned}$$

We have to determine the actual C^i_x, C^i_{dp} and C^i_{tp} parameters for each task-subroutine. Consider equation (1). By executing a task-subroutine three times on different number of processors, and measuring the execution time, we obtain three equations in three unknowns, which can be solved in a straightforward way to obtain all C^i_x values. Similarly, by measuring the data transfer time for two different executions for equation (2), and three different executions for equation (3), the unknown parameters can be determined by solving systems of linear equations.

We omit the details of the derivation of memory requirement parameters, and simply state that they can be inferred using compile time information and measurements of stack and heap allocations, made during execution with two different mappings.

Lemma 1 *All the parameters of the execution model described in Section 3 can be obtained by executing a program at most 5 times.*

Proof: The parameters for data parallel and task parallel communication can be obtained (for instance) by executing the program in pure data parallel mode twice and in pure task parallel mode three times, that is 5 distinct executions. The parameters for execution time and memory requirement each require two different executions and can be included in the set of 5 executions stated above.

We wish to point out that the final parameterized equations only approximate the real behavior of the program. In practice, however, they provide sufficient information for the mapping process.

5 Building modules from tasks

Once the parameters of the execution model for a program are established, it is possible to predict the total execution time for any program mapping. The objective is to find the optimal mapping for a given program on a given set of processors. To establish the mapping of a program onto a set of processors, the following decisions have to be made:

1. Partitioning of task-subroutines into modules.

2. Possible replication of modules.

3. Allocation of processors to module instances.

In this section, we address the problem of partitioning task-subroutines into modules. We initially place every task-subroutine in a module by itself, and then selectively merge pairs of modules, whenever it is considered profitable. We restate that all task-subroutines in the same module are mapped to the same set of processors.

One reason why the mapping problem is difficult is that the three subproblems listed above are interdependent, that is, the optimal solution for each depends on the solutions of others. We present a set of results that address the problem of partitioning tasks into modules, with some assurance that the final mapping obtained from this partitioning is close to optimal. In the next section, we address replication of modules and processor allocation.

5.1 Theoretical complexity

It is fairly easy to show that the problem of optimally mapping a set of parallel tasks to a set of processors is in a class of multiprocessor scheduling problems that are NP-hard. For this purpose, we consider a parallel machine with only two processors, and assume that there are no memory or dependence constraints. The restricted optimization problem obtained can be stated as follows:

Given a set of tasks $T_1, T_2, \ldots T_n$, with execution time $\tau(i)$ for task T_i. Divide the tasks into two groups such that the maximum of the sum of the two execution times in the two sets is minimized.

This problem can be shown to be NP-hard by transforming the *Partition* problem [7] to the above problem.

5.2 Pair of tasks

We first consider a pair of task-subroutines that are assigned to different modules, and examine the performance implications of merging the modules. There are several reasons why it may be profitable to assign a pair of task-subroutines to *different* modules:

- More processors may be used effectively by two different modules than one.
- Replication and processor assignment decisions for the task-subroutines are decoupled, allowing more flexibility.
- Individual memory requirements for each module may be lower than the memory requirement of a single merged module, allowing each instance to fit in a smaller number of processors and improving communication locality.

However, the cost of transferring data between task-subroutines can increase when they belong to different modules, and are mapped on different parts of the processor array. In particular, when the distribution of an array item is the same for the two task-subroutines, the data transfer cost is zero when they belong to the same module, but can be significant when they are in different modules.

It is possible to make the decision on merging the modules in a provably correct way, if the following condition holds for the corresponding task-subroutines:

1. The task-subroutine scales linearly, including inter-task communication cost, *or*

2. The task-subroutine can be replicated, and the total number of processors available is much greater than the minimum number needed to map the subroutine.

This condition allows us to assign fixed execution speed (per processor), with task-subroutines. If a task-subroutine scales perfectly, then this is trivial. Otherwise, we pick the fastest feasible execution rate, which corresponds to execution with the *smallest* number of processors that the task-subroutine can execute on, since that provides the best communication locality. Under this condition, the following lemma holds:

Lemma 2 *Consider a program consisting of two task-subroutines T_1 and T_2, which can be mapped to individual modules M_1 and M_2, or a single module M_{12}. Let the* minimum *number of processors required to fit the data sets of corresponding modules be P_1, P_2 and P_{12}. The criterion for placing both the task-subroutines in the same module is:*

$$(P_1\tau_x^1 + P_2\tau_x^2)/(P_{12}\tau_x^{12}) > 1$$

where τ_x^1, τ_x^2 and τ_x^{12} are the execution times (including communication) of the corresponding modules, on P_1, P_2 and P_{12} processors, respectively.

The result is obtained by comparing the predicted speeds of execution for the two cases, and is our basis for dividing task-subroutines into modules. The condition is verified in a quantitative sense, and in our experience, usually holds. An iterative approach is used when the lemma cannot be used directly.

5.3 Task graphs

We now address the problem of partitioning a task graph, which is required to be acyclic, except for self dependences. The result from last section can be used to make pairwise decisions, but the order in which the pairs are chosen can influence the final partitioning. Consider the task graph in Figure 4. Task-subroutines T_1, T_2 and T_3 require a minimum

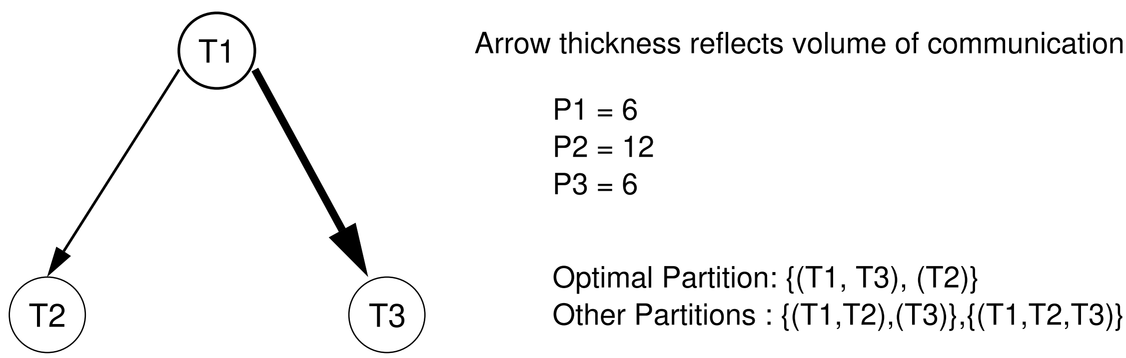

Figure 4: Partitioning a task graph into modules

of 6, 12 and 6 processors, respectively, to execute. For simplicity, we assume that the minimum number of processors required to execute a module is the maximum of the processors required for individual subroutines inside the module. Also, suppose T_3 is a communication intensive task-subroutine that does not scale well, while others scale linearly, and the volume of communication along the edge T_1->T_3 is the only major inter-task communication cost, which is non-existent if those two task-subroutines are mapped to the same module.

The optimal partitioning into modules is $\{(T_1, T_3), (T_2)\}$, which is obtained if we examine T_1 and T_3 first, and merge them into a module. However if we examine T_1 and T_2 first, we may merge them, and then the possible partitionings are $\{(T_1, T_2), (T_3)\}$, which implies that the cost of communication from T_1->T_3 has to be paid, or (T_1, T_2, T_3) which implies that T_3, that does not scale well, will be forced to use at least 12 processors per instance instead of 6, which is wasteful.

In general, the problem is that we may combine a set of compute intensive routines in one module first (with limited benefit), which may leave a communication intensive routine with limited choices. Based on these observations, we use the following heuristic to pick a pair of modules for a potential merger:

- Higher priority to modules that do not scale well as number of processors is increased.
- Higher priority to pairs of modules that need roughly the same number of processors, since the decision influences other decisions less.

The pairwise selection method discussed here is effective for task graphs that are trees and our tool set uses it. We are motivated by the fact that a large class of applications lead to task graphs that are trees, often just straight line graphs. There is some additional complexity involved in the partitioning of acyclic task graphs that are not trees, and we expect to address that in future publications.

An obvious alternate approach is to exhaustively try all orders and use the best results obtained. Although this approach does not seem appealing, it can be used effectively in many situations, since task graphs in a data and task parallel program often contain only a few nodes.

6 Mapping modules to processors

Once the task-subroutines have been partitioned into modules, we divide the processors among the modules, and *replicate* the modules when legal and profitable. Replication means executing multiple instances of modules on different parts of the processor array, and is legal when none of the task-parameters in the module has a self dependence, or carries *state* between instances. The multiple module instances share work by processing data-sets in a round robin fashion. Our procedure is as follows:

1. Divide the processors among modules based on the best estimate of their execution rate.
2. Map the modules onto the processor array, replicating whenever it is possible and profitable.

6.1 Allocating processors to modules

The objective of processor allocation is that every module should execute in approximately the same amount of time, thus minimizing load imbalance. The execution time of a module containing a set of task-subroutines, including the communication time, on P processors, can be modeled as:

$$\tau_x^M(P) = C_x^0 + (C_x^1/P) + (C_x^2 * P) + \sum C_x^i * P_i$$

where each term in the summation refers to communication with another module. The parameters can be computed directly from the parameters of the task-subroutines that constitute the module. The summation term involves the number of processors allocated to other modules that this module communicates with. On per processor basis, the best performance is achieved when an instance of the module executes on the smallest number of processors. Thus, if each instance of the module requires at least P_0 processors to execute, and the modules that it communicates with are also assumed to execute on the smallest set of processors (P_{i0}), the execution time corresponding to the best per processor performance is:

$$\tau_x^M(P_0) = C_x^0 + (C_x^1/P_0) + (C_x^2 * P_0) + \sum C_x^i * P_{i0}$$

which is a constant. And the execution rate of module M is:

$$\mathcal{X}(M) = 1/(\tau_x^M(P_0) * P_0)$$

in datasets/second/processor.

Once fixed per processor execution rates are assigned to modules, the problem of finding an optimal allocation can be stated as:

Given a set of modules $M_1, M_2, \ldots M_k$, with execution rate $\mathcal{X}(M_i)$ for module M_i, divide the available processors P into k sets $P_1, P_2, \ldots P_k$, such that $max_{i=1,k}(P_i * \mathcal{X}(M_i))$ is minimized.

We first solve the problem exactly assuming P_is are allowed rational, i.e. non integral values. We assign an arbitrary value to (say) P_0, then solve for $P_1, P_2, \ldots P_k$ using $P_i * \mathcal{X}(M_i) = P_0 * \mathcal{X}(M_0)$ and scaling all P_i values such that $\sum_{i=1}^{k}(P_i) = P$. Actual numbers of processors assigned to each module, of course, must be integers. Once a rational solution is known, we simply find an approximate integer solution that is close to the rational solution.

As in the last section, the processor assignment procedure is based on finding a fixed, per processor, execution rates for modules. When this is not possible, an iterative approach is used.

6.2 Replication and processor assignment

As stated before, replication of a module is legal if none of the task-subroutines in it has a self cycle. Even when replication is legal, it may not be feasible or desirable for the following reasons.

- Enough processors may not be available.
- When the communication paradigm requires reservation of machine resources (e.g. long lived connections in some parallel machines and networks), sufficient resources may not be available.
- A large number of module instances increase the possibility of slowdown due to sharing of communication resources.

We make the actual replication decisions heuristically, taking the above factors into account. We omit the details, but typically the result is that the communication intensive routines that cannot use a large number of processors in one instance effectively, are more likely to be replicated, while subroutines that scale well are less likely to be replicated.

Finally, we have to assign the physical processors to module instances. This is done such that the sharing of communication resources between modules is minimized. In our current implementation, module instances can be mapped only to rectangular subarrays of processors.

It may be useful to re-execute the program with the predicted mapping, and collect information for fine tuning. We expect to be able to comment on the importance of this step with more experience.

7 Application to an example program

We illustrate the process of automatic mapping by demonstrating how our toolset maps an example program onto a 64 processor iWarp array. The program consists of a 512 point 2DFFT, implemented as a sequence of 1DFFTs with a transpose between them, followed by a statistical analysis routine. The three task-subroutines corresponding to the computation stages are `colffts`, `rowffts` and `hist` (for column FFTs, row FFTs and Histograms), respectively. Figure 5 shows the structure of the computation, and a task-subroutine call level code for the program.

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

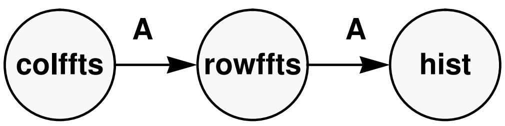

Figure 5: Structure of the example program

The first two task-subroutines are completely parallel, while the third task-subroutine, `hist`, contains significant communication. This example was chosen because it demonstrates the tradeoffs between different mapping styles, and more important, it reflects the structure of a large class of applications in digital signal processing and image processing.

7.1 Obtaining execution parameters

The program is executed for a set of different mappings, and the instrumentation generates timing information for the execution of the task-subroutines, and for communication between them. The mappings used for this example are shown in Figure 6. In accordance with Lemma 1, these 5 mappings are sufficient to derive the parameters of the program model presented in section 3. The measured execution and communication times are tabulated in Figure 7.

Analysis of these measurements using the methods discussed in section 4, yields the execution parameters of the program. The execution times of the task-subroutines in terms of the number of processors are as follows:

$$\begin{aligned}
\tau_x^1(P) &= -0.97 + (0.011 * P) + (600.4/P) \\
\tau_x^2(P) &= -6.42 + (0.072 * P) + (639.0/P) \\
\tau_x^3(P) &= 15.27 + (0.231 * P) + (252.2/P)
\end{aligned}$$

The estimated execution time function is plotted in Figure 8 and shows that the FFT stages scale very well, while the histogram stage does not. This reflects the computations in the task-subroutines - the first two are data parallel, while the third is communication intensive.

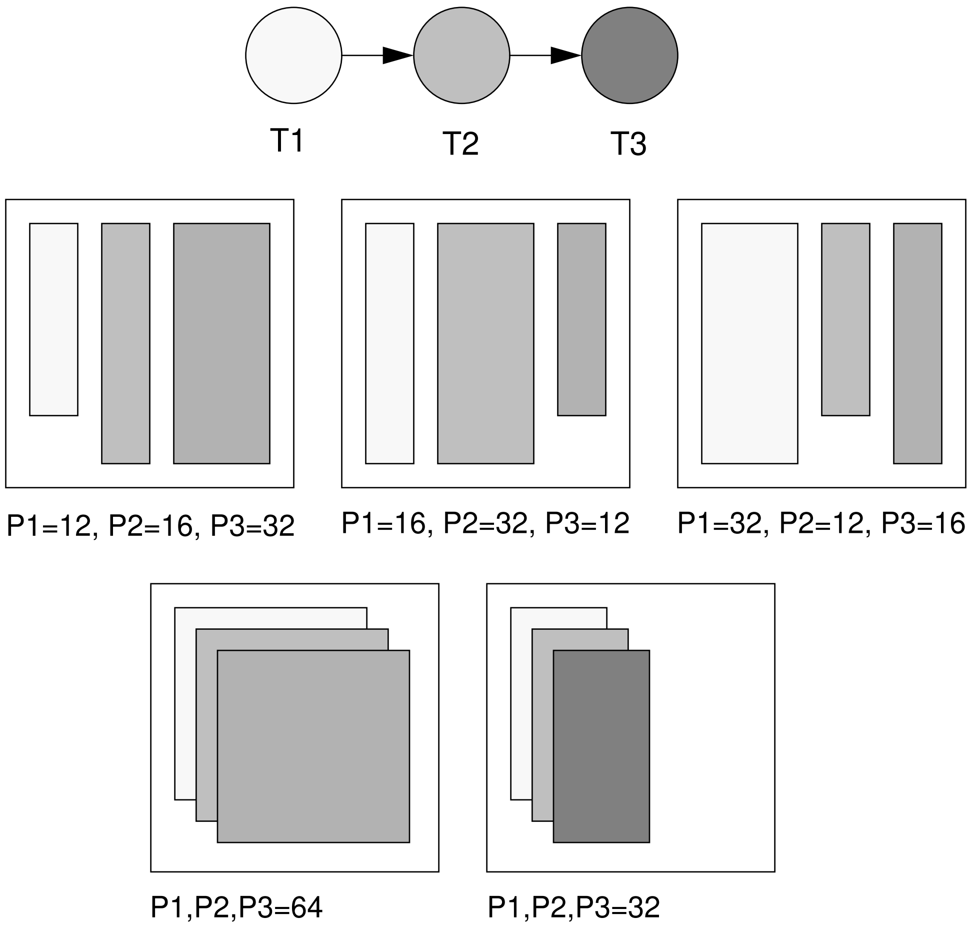

Figure 6: Mappings for analyzing the example program

Processors	Execution Time (10^{-2}*secs.*)		
	T1	T2	T3
12	49.18	47.67	39.06
32	18.12	15.82	30.54
64	9.11	8.16	34.00

Processors (send-task -> recv-task)	Communication Time (10^{-2}*secs.*)			
	Data Parallel		Task Parallel	
	T1->T2	T2->T3)	T1->T2	T2->T3
32->32	4.18	0.08	*	*
64->64	2.50	0.06	*	*
12->16	*	*	9.57	9.61
16->32	*	*	9.65	9.63
32->12	*	*	9.80	9.95

Figure 7: Timings from evaluation runs of the example program

The communication parameters, also derived from the measurements in Figure 7, are as follows:

$$\tau_{dp}^{1->2}(P) = 5.86 - (0.052 * P)$$
$$\tau_{dp}^{2->3}(P) = 0.10 - (0.001 * P)$$

$$\tau_{tp}^{1->2}(P_1, P_2) = 9.39 + (0.012 * P_1) + (0.002 * P_2)$$
$$\tau_{tp}^{2->3}(P_2, P_3) = 9.46 + (0.016 * P_2) - (0.003 * P_3)$$

We note that there is some cost of data parallel communication between task-subroutines T_1 and T_2, but practically

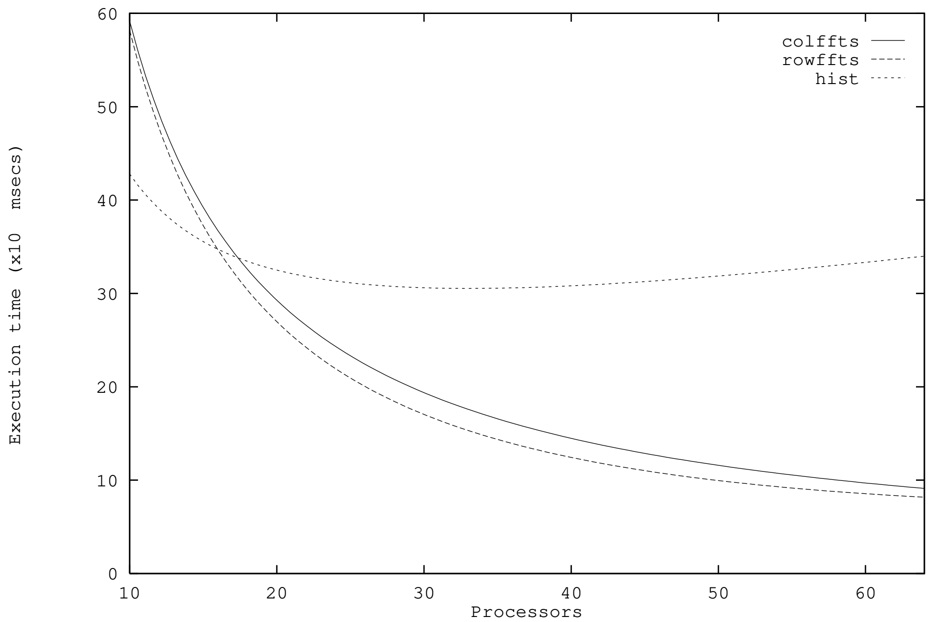

Figure 8: Scaling of example program subroutines

none between T_2 and T_3. This is because the former involves a matrix transpose, while the latter requires no data movement. The task parallel communication time is practically constant. In our implementation, the communication between tasks is done systolically on a single link with a fixed bandwidth, and hence the communication time primarily depends on the volume of data being transferred, which is the same in all cases.

We omit the details of the memory requirement analysis and state the final results. A processor in our target machine (iWarp) has a fixed amount of main memory. The minimum number of processors required to fit the data set of each of the three task-subroutines is 10, which is the same since the main data structure is the same array. However, when task-subroutine T_1 is mapped to the same module as T_2 or T_3, a minimum of 20 processors are required, since memory for the original and the transposed array must be allocated.

7.2 Modules from tasks

We initially put each task-subroutine in a module by itself, and compare pairs of modules that have a communication edge between them, to determine if the modules should be merged. In the example program, we have modules M_1, M_2 and M_3 containing the three task-subroutines. Using the priority system discussed in 5.3, we first examine the pair (M_2, M_3), since M_3 contains the least scalable task-subroutine. Using Lemma 2, we compute:

$$(P_2\tau_x^{M2} + P_3\tau_x^{M3})/(P_{23}\tau_x^{M23}) = 1.189 > 1$$

where $P_2, P_3 = 10$, $P_{23} = 10$ and execution times are computed from the equations for the corresponding modules. Since the above expression > 1, we conclude that the modules should be combined, yielding a new module M_{23}. The execution parameters of M_{23} are obtained by adding the execution parameters of P_2 and P_3 and the data parallel communication parameters for communication between them. We obtain:

$$M_x^{23}(P) \quad = \quad 8.95 + (0.302 * P) + (891.2/P)$$

As before, we test if the modules M_1 and M_{23} should be merged and obtain:

$$(P_1\tau_x^{M1} + P_{23}\tau_x^{M23})/(P_{123}\tau_x^{M123}) = .945 < 1$$

and therefore the modules are not combined. Thus the final assignment of task-parameters to modules is:

$$\begin{aligned} M_1 &= \{(T_1)\} \\ M_{23} &= \{(T_2, T_3)\} \end{aligned}$$

7.3 Allocation of processors

The processors are allocated on the basis of the relative execution speed for the minimum number of processors that the modules run on, including the communication time. In this case, we have:

$$(\tau_x^{M1} + \tau_{tp}^{M1})/(\tau_x^{M23} + \tau_{tp}^{M23}) = 1 : 1.57$$

And for a 64 processor machine, we get

$$\begin{aligned} P_1 &= 24.9 \approx 25 \\ P_{23} &= 39.1 \approx 39 \end{aligned}$$

7.4 Replication

The replication decisions are based on the scalability of different modules - the less scalable are prioritized. We measure the execution time for running each module on all the processors allocated to it, and with only half of the processors allocated, and pick the module which shows the least speedup. For our example, we get:

$$\begin{aligned} \tau_x^{M1}(12.5)/\tau_x^{M1}(25) &= 2.02 \\ \tau_x^{M23}(19.5)/\tau_x^{M23}(39) &= 1.39 \end{aligned}$$

Hence module M_{23} is selected for replication. It has 39 processors allocated, and each instance needs at least 10, so we overallocate, and have four instances, each executing on 10 processors, consuming 40 processors. The remaining 24 processors are allocated to module M_1. Since module M_1 shows nearly linear speedup, it is not replicated.

Figure 9 summarizes the steps in mapping the example program, and shows the final mapping. Figure 10 compares the performance of this mapping to other mappings.

8 Discussion

We are using the results of this research for developing several applications with the Fx compiler. We obtained four fold improvement over a fully data parallel Narrowband tracking radar benchmark [11], mainly because the data parallel version could use only a small number of cells effectively. Other applications under development include SAR (Synthetic Aperture Radar), Multibaseline Stereo, and MRI (Magnetic Resonance Imaging). While the discussion of these applications is beyond the scope of this paper, they all have multiple stages with different computation requirements, similar to the model program discussed in section 7.

The main limitation of our approach is that it can be used only for programs where the history of execution is a good indicator of the future computation and communication requirements. In the current implementation, the mapping is fixed during execution, and cannot change with runtime behavior. While our approach is not applicable to programs whose runtime behavior is completely unpredicatable, it can be adapted for programs where the runtime behavior can change, but recent history is still a good predictor of near future. We are planning to develop an implementation that supports dynamic remapping based on changes in runtime behavior. While this will add considerable complexity to the system, the fundamental approach remains the same.

We have selected a fairly simple execution model and many refinements are possible. We model the execution time as a quadratic functions, and the communication times as linear functions. We also ignore several secondary effects, for instance, the effect on performance due to sharing of the global machine resources. Our guiding principle has been to develop a simple model and implementation that can be used effectively and conveniently for developing applications. We expect experience to guide us into refinement of the automatic mapping tool.

Plans for future research include development of an interactive tool to guide program mapping, and to port the implementation to other parallel computers and high speed networks. We are also in the process of identifying more applications which can profit from an integrated task and data parallel compiler.

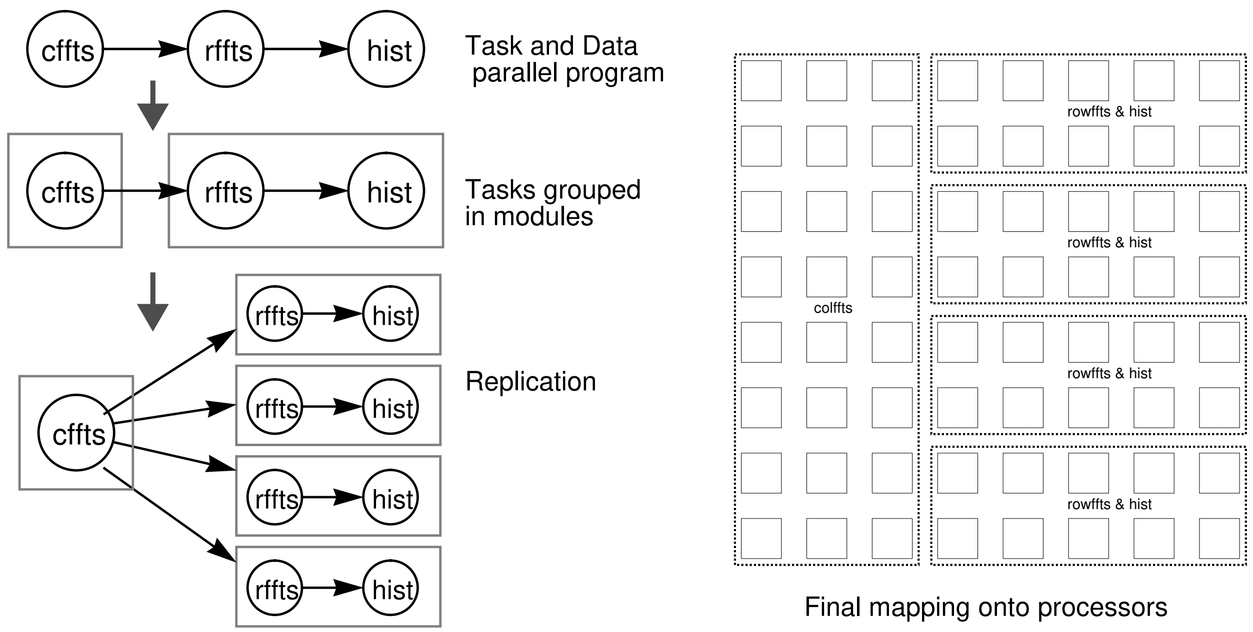

Figure 9: Mapping steps and the final mapping of the example program

Program Mapping	Performance (data-sets/sec.)
Data Parallel	1.44
Best Task Parallel	2.06
Best Predicted (as shown)	2.82

Figure 10: Relative performance of the final mapping

9 Related Work

Compilation and optimization of programs for private memory parallel computers has been a very active area of research for several years. Several parallelizing compilers have been developed for data parallel programs, including Fortran D [13] and Vienna Fortran [3], and for task parallel programs [8, 9]. Recent research shows that a large class of applications contain task and data parallelism [6] and it is important to exploit them in a single compiler framework [4, 5, 12]. There is also a large body of literature on, partitioning, load balancing and scheduling of parallel programs [1, 10].

We have addressed the specific partitioning and load balancing issues that arise when task and data parallelism are combined in a parallelizing compiler, including memory requirement issues that are important but often ignored, and developed a practical system to efficiently compile and map task and data parallel programs. An alternate approach, taken in Jade [8] is to express all parallelism as coarse grain tasks, and make scheduling decisions at runtime. This is particularly useful when runtime behavior is unpredictable, but may entail a higher overhead. Most applications have components that have simple data parallelism, and we believe that it is extremely important to use an optimizing data parallel compiler for integrated task and data parallel systems, and statically schedule and optimize computations and communication whenever feasible.

10 Conclusions

We have examined the fundamental characteristics of task and data parallel programs that determine their performance, and presented an execution model for such applications. We used this model and analysis techniques to build a tool to automatically map parallel programs onto a parallel machine. We are using this tool to develop a class of applications that need to exploit task and data parallelism for efficient execution.

References

[1] BOKHARI, S. *Assignment Problems in Parallel and Distributed Computing*. Kluwer Academic Publishers, 1987.

[2] 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.

[3] CHAPMAN, B., MEHROTRA, P., AND ZIMA, H. Programming in Vienna Fortran. *Scientific Programming 1*, 1 (Aug. 1992), 31–50.

[4] CHEUNG, A., AND REEVES, A. Function-parallel computation in a data-parallel environment. In *Proceedings of the 1993 International Conference on Parallel Processing* (St Charles, IL, August 1993).

[5] FOSTER, I., AND CHANDY, K. Fortran M: A language for modular parallel programming. Tech. Rep. MCS-P327-0992, Argonne National Laboratory, June 1992.

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

[7] GAREY, M., AND JOHNSON, D. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Fransisco, 1979.

[8] 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).

[9] PRINTZ, H. *Automatic Mapping of Large Signal Processing Systems to a Parallel Ma chine*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991. Also available as report CMU-CS-91-101.

[10] SARKAR, V. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. The MIT Press, Cambridge, MA, 1989.

[11] SHAW, G., GABEL, R., MARTINEZ, D., ROCCO, A., POHLIG, S., GERBER, A., NOONAN, J., AND TEITELBAUM, K. Multiprocessors for radar signal processing. Tech. Rep. 961, MIT Lincoln Laboratory, Nov. 1992.

[12] SUBHLOK, J., STICHNOTH, J., O'HALLARON, D., AND GROSS, T. Exploiting task and data parallelism on a multicomputer. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (May 1993).

[13] TSENG, C., HIRANANDANI, S., AND KENNEDY, K. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93* (Portland, OR, November 1993).