Communication and Memory Requirements as the Basis for Mapping Task and Data Parallel Programs

Jaspal Subhlok David R. O'Hallaron Thomas Gross
Peter A. Dinda Jon Webb

August 1994
CMU-CS-94-106 R

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Appears in the Proceedings of Supercomputing '94

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. The tradeoffs between task and data parallelism are complex and depend on the characteristics of the program to be executed, most significantly the memory and communication requirements, and the performance parameters of the target parallel machine. In this paper, we present a framework to isolate and examine the specific characteristics of programs that determine the performance for different mappings. Our focus is on applications that process a stream of input, and whose computation structure is fairly static and predictable. We describe three such applications that were developed with our compiler: fast Fourier transforms, narrowband tracking radar, and multibaseline stereo. We examine the tradeoffs between various mappings for them and show how the framework is used to obtain efficient mappings.

This research was sponsored by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152, and also by the Air Force Office of Scientific Research under contract F49620–92–J–0131.

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

1 Introduction

Parallelizing compilers typically support data parallelism [7, 25] or task parallelism [16, 19], but not both. However, many applications contain a mix of task and data parallelism [1, 6, 10, 12], and it is important for the compiler to exploit both styles [8, 11, 24]. In this paper, we address how program characteristics influence the tradeoffs in task and data parallel programs, and present a framework for finding good parallel mappings.

There are several reasons to support both task and 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, if the program does not scale well. A complete application often consists of a set of procedures with different computation and communication requirements, and different sets of processors may have to be assigned to different procedures for the best possible performance.

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 replicated 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).

The fundamental question addressed in this paper is: "Which of the many mappings that are feasible when task and data parallelism are exploited together is optimal, and how can it be determined?" To address this question, we examine three critical measurable characteristics of task and data parallel programs: scalability, memory requirement and inter-task communication cost. We build a framework to methodically choose a task and data parallel mapping based on these program characteristics. The central idea is to find the best use of available memory and communication resources to minimize the global inter-task and intra-task communication overhead. We show how the framework is used to map three applications in signal and image processing: fast Fourier transforms, narrowband tracking radar and multibaseline stereo imaging, and we present performance results.

2 Programming and compiling task parallelism

Our model for combining task and data parallelism in the same program forms the foundation of the Fx compiler system [24]. The base language is Fortran 77 augmented with Fortran 90 array syntax and data layout statements based on Fortran D[25] and High Performance Fortran [15]. Data parallelism is expressed using array syntax and parallel loops. The main concepts in the Fx compiler are language independent. Fortran was chosen for convenience and user acceptance.

Task parallelism is expressed in Fx using special code regions called *parallel sections* [14]. 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 generate 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.

The compiler can map and schedule instances of task-subroutines in any way, as long as sequential execution results are guaranteed. 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 and each module runs on a unique set of processors. The modules may be *replicated* to generate multiple module instances, with each instance receiving and sending data sets in round robin fashion. For example, if a module containing a single task-subroutine is replicated to two module instances, the task-subroutine will be mapped to two distinct sets of processors, with each set executing alternate calls to the subroutine. The idea of replication is not common in the mapping literature. However, replication is extremely useful for modules that process a stream of independent inputs, have no carried state, and do not scale well. This is typical of signal and image processing applications. Constraints on replication are discussed in Section 5.3. Finally,

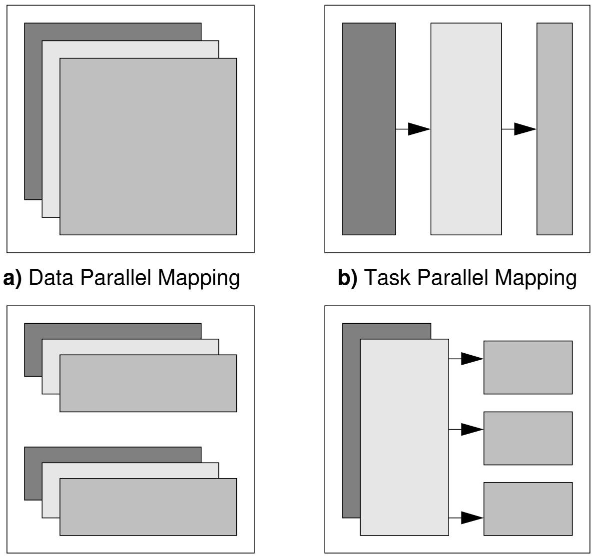

Figure 1: Combinations of Task and Data parallel mappings

```
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
```

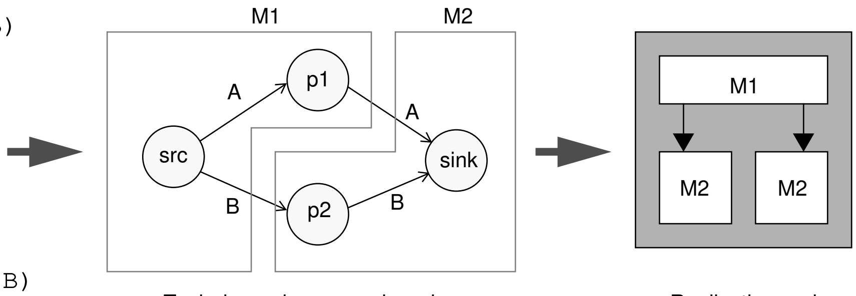

Figure 2: Compilation of task parallelism

a subset of the processor array is assigned to each module instance. All mapping decisions are controlled by the compiler, possibly using other tools.

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 clustered into modules M1 and M2. Module M2 is replicated to two instances, and the program is mapped onto the machine.

Our approach to task parallelism is different from Fortran M [11] in the sense that data and task parallelism are strongly integrated; both are expressed and compiled uniformly using a sequential Fortran program and additional directives. (However, an effort is underway to integrate task and data parallelism more tightly in Fortran M [5]). Our approach is similar to the OPUS language, where data parallel tasks communicate and synchronize via *shared data abstractions* [6], which are generalizations of Fortran 90 modules that combine properties of objects and monitors. However, in our case, the shared objects are simple arrays, and interaction between tasks is limited to procedure boundaries. In general, our approach favors simplicity and efficiency over generality of expression.

Another approach to task parallelism, taken by Jade [16], is to express all parallelism as coarse grain tasks, and make scheduling decisions at runtime. Most applications have components that have simple data parallelism, and it is important to use an optimizing data parallel compiler for integrated task and data parallel systems as well as to statically schedule and optimize computations and communication whenever feasible.

3 The mapping problem

The mapping framework discussed in this paper is applicable to programs that contain a set of task-subroutines that process a long stream of inputs in pipelined fashion. The task-subroutines are already partitioned by the data parallel compiler to run on a variable number of processors, and their parallel behavior is characterized by the following:

- *Response time function*: The execution time of a task-subroutine call as a function of the number of processors.
- *Communication time function*: The time to transfer data between a pair of task-subroutines as a function of the number of processors executing the subroutines, and whether or not the pair of task-subroutines is on same or different sets of processors.
- *Memory requirement function*: The memory requirement on each processor for a task-subroutine as a function of the number of processors.

These characteristics are assumed to be independent of the values of the input data set and known at compile time. They can be determined experimentally or analytically. In related research, we have shown that a small number of program executions are sufficient to determine them with sufficient accuracy for the purpose of identifying a good mapping for the programs in the domain of our mapping framework [23].

The problem that we are solving in this paper is how to best use the available processors to maximize the *throughput* of the system. This problem is fundamentally different from the many partitioning problems addressed in the literature (e.g., [3, 17, 21]) for several reasons: (1) the task-subroutines are to be mapped to groups of processors, not individual processors, (2) the computation and communication costs are functions of the number of processors, not constants, and (3) the objective is to maximize throughput, not to minimize execution time.

More recently, the problem has been addressed by other researchers, but not in its general form. Choudhary et. al. [9] discuss the processor allocation problem, but do not address clustering or replication and do not take into account communication costs that are not part of the response function. Cheung and Reeves [8] address replication for a single task-subroutine, but not clustering. The Paradigm compiler [20] attempts to minimize execution time for a task graph, but the problem is different from maximizing throughput for a stream of inputs. We also take memory requirements into account, which are not considered by others, but are of fundamental importance in using a parallel machine.

It is necessary to address the more general problem to be able to develop real applications with a parallelizing compiler, even though the complexity of the problem forces a more heuristic approach. Computer vision, image processing, and signal processing are examples of application domains where many programs fit our model, and pure data parallel computing often leads to disappointing results. In this paper we present a methodology for mapping task and data parallel programs that can be used automatically, or by the programmer with a profiling tool.

4 Characteristics of parallel programs

A parallel program in our language contains a set of data parallel task-subroutines, with communication between them. We identify three measurable properties of parallel programs based primarily on communication characteristics and sizes of data structures.

1. Scalability.
2. Inter-task communication.
3. Memory requirement.

The rest of this section describes these characteristics, which will serve as the basis of the mapping framework presented in the next section.

4.1 Scalability

As the number of processors allocated to a fixed size computation (or a task) is increased, the average processor efficiency often decreases. There are a variety of program characteristics that influence scalability, particularly the communication overhead and the size of distributed data structures. We identify four types of programs and show their scalability profile in Figure 3. Plot A represents a perfectly scalable parallel program with no communication. Plot B represents a parallel program with communication, where average processor efficiency decreases gradually due to increasing communication overhead, as the number of processors increases. Plot C represents a purely sequential computation which can use only one processor. Plot D represents a computation which scales perfectly up to a fixed number of processors, but cannot use any more processors. In general a program represents a mix of these behaviors.

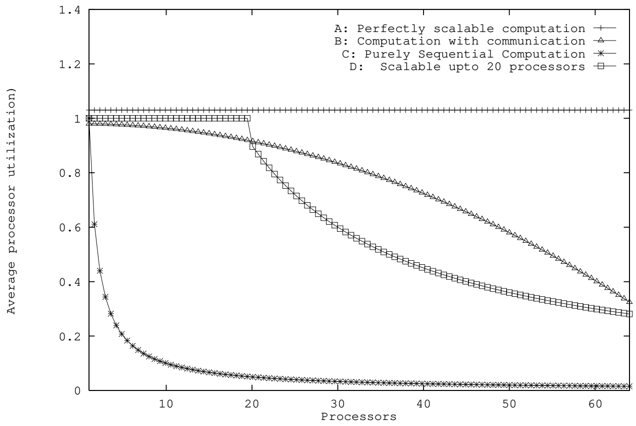

Figure 3: Scalability of computations

4.2 Communication between task-subroutines

The volume of data to be transferred in a communication step between a pair of task-subroutines is a feature of the program, but the actual cost of a communication step depends on several other factors: the number of processors executing the task-subroutines, the data distributions, and the communication style used, which is determined by whether the two task-subroutines are in the same or different modules.

Number of processors

The cost of communication depends on the number of processors assigned to the corresponding task-subroutines. For communicating a fixed amount of data, the per processor efficiency of communication decreases with increasing number of processors, while the total communication time can increase or decrease.

Data distributions

In general, the granularity, and hence the overhead of communication, is dependent on the data distributions. When the communicating task-subroutines belong to the same module, the cost varies from zero (if the distribution of data is the same in both subroutines) to the cost of a fine grained all-to-all communication (if, e.g., one distribution is a transpose of the other).

Communication style

When a pair of task-subroutines is in the same module (data parallel style), the data exchange is a local redistribution of data, and when in different modules (task parallel style), the data is sent from one set of processors to another set. These two are different problems, and the cost characteristics can be different even if the communication patterns are similar. In the implementation for iWarp discussed in the examples of this paper, inter-module communication is done systolically over a single channel, which has an extremely low overhead, but limited bandwidth. Intra-module communication is based on a message passing library, which has a relatively higher overhead, but more parallelism and bandwidth.

4.3 Memory requirements

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 minimum number of processors needed to execute a task-subroutine or a module containing a set of task-subroutines. In the presence of a memory hierarchy, the memory requirement plays an important role in determining overall performance.

5 A framework for mapping programs

In this section, we outline a framework for reasoning about the mapping of task and data parallel programs onto multicomputers. The objective of program mapping is to maximize throughput. During execution, a processor is in one of the following states:

- Executing a task-subroutine
- Communicating between task-subroutines
- Idling due to load imbalance

In general, the most important criterion for high throughput is efficient execution of task-subroutines. This may require that the number of processors allocated to individual computations should be small, if the subroutine does not scale well. However, reducing the number of processors assigned to a computation may not be profitable, or even feasible, due to other factors, e.g. memory requirement. We state the main steps in the process of program mapping, and discuss the tradeoffs that have to be made at each step. Following are the major steps in deriving an efficient mapping for a parallel program expressed as data parallel task-subroutines:

Step 1: Cluster task-subroutines into modules.

Step 2: Allocate processors to modules.

Step 3: Replicate modules into module instances.

The rest of this section presents a general framework to make these decisions in the above order, although the steps are dependent. In particular, we have to assess the impact of replication before we make a clustering decision.

5.1 Clustering

The first step is to cluster the task graph obtained from the program into modules. The nodes in the task graph are data parallel procedures which can be potentially replicated, and whose cost is a function of the number of processors assigned to them. The edges correspond to communication cost, which varies with the number of processors, and also depends on whether the corresponding nodes are assigned to the same or different modules.

We first assign a module to each task-subroutine, and compare pairs of neighboring modules to determine if they should be merged into a single module. The order in which pairs of modules are tested for potential merger is determined heuristically based on the following priority scheme:

- Less scalable modules are prioritized, since their mapping is most important for efficient execution of the program.
- A pair of modules that share most of the data structures is prioritized, since a decision to combine them has minimal impact on other decisions.

We now discuss the procedure for deciding if a pair of modules should be merged or not. We first examine the considerations for this decision:

Replicability: If one of the modules cannot be replicated, merging them implies that neither of them can be replicated, hence the unmerged modules only will benefit from replication.

Memory constraints: Merging the modules increases the memory requirement, and hence the minimum number of processors that the merged module needs to execute can be higher than the minimum number of processors needed by separate modules. This can restrict (or prevent) replication, and limit the best performance, as illustrated in Figure 3.

Inter-task communication cost: If the task-subroutines are merged, the communication step is a data redistribution, and can be zero if the data structures have the same distribution in both task-subroutines. If they are not merged, the data must be transferred between sets of processors.

We now state the specific criterion for making this decision assuming replication is possible for both modules. The obvious approach is to estimate the performance in the two situations, but the difficulty is that the performance of non-scalable modules depends on the number of processors assigned to each module, and that is not known. However, when the modules can be replicated, an optimistic estimate of performance can be obtained by assuming that each instance of the module will be assigned the minimum number of processors that are necessary due to memory or I/O requirements. So, if we are considering merging modules M_1 and M_2 into a single module M_{12}, and the *minimum* number of processors required to fit the data sets of corresponding modules is P_1, P_2 and P_{12}, we use these processor numbers to estimate computation and communication performance.

5.2 Processor allocation

The allocation of processor is based on the execution time estimates of the modules discussed in the last section. Methods discussed in [9] can be used when fixed estimates are not possible.

5.3 Replication

Replication improves the performance by reducing the execution overhead of modules, as well as the cost of communication between them, since each instance executes on a smaller number of processors. The effect is most pronounced for modules that do not scale well. However, it is not possible to arbitrarily replicate modules into smaller module instances. The legality and the degree of maximum possible replication is determined by the following constraints:

Dependence Any module that has a carried dependence (or carries *state* across executions) cannot be replicated.

Memory requirement Every module needs a minimum number of processors to hold the data set, which bounds the number of module instances possible for a fixed set of processors.

Latency The latency between receiving an input and generating an output can increase with replication, even if the throughput increases. Although we have assumed that our objective is to maximize throughput, the extent of latency that can be tolerated may be limited in real-time applications.

Resource constraints Supporting a finer granularity of task parallelism obtained by replication increases the demand for machine resources. This can constrain replication either because resources are not available (e.g. communication channels) or increase the overhead due to congestion. Communication resources also restrict the number of modules that can interface with external I/O agents.

For the purpose of determining a clustering, we assume maximum replication allowed by the constraints stated above, but do not consider the effect of resource consumption of one module on the availability of resources for others. This is done to obtain computation and communication time estimates before clustering into modules is complete.

The actual replication is done in the order of increasing scalability, that is, the least scalable modules are replicated first. For each module, we pick the minimum number of processors that are needed to obtain nearly linear speedup, and replicate until that number of processors per instance is reached, or one of the constraints prevents further replication. The process stops when all the modules are processed, or one of the essential resources is exhausted.

6 Example programs

In this section, we apply the framework developed in the previous section to three realistic task and data parallel programs: fast Fourier transform, narrowband tracking radar [22], and multibaseline stereo imaging [28]. We briefly describe each example program, show how it is implemented as an Fx program, and show how to use the framework to reason about mapping task parallel programs. We present performance results on a 64 processor iWarp array [4] to validate our approach.

Each example program has unique properties that exercise different parts of the mapping framework. Each program operates on a stream of input, and generates a stream of output. While the I/O issues involved are beyond the scope of this paper, we assume that the I/O streams can be partitioned into up to four interleaved I/O substreams.

6.1 Fast Fourier transform

The fast Fourier transform (FFT) is of fundamental importance in signal and image processing. The FFT is interesting to study because its task parallel structure is similar to many larger signal processing applications, and because it is a satisfying combination of the easy and the challenging: easy because there is perfect data parallelism within each task, 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 FFT is an efficient procedure for computing the discrete Fourier transform (DFT). A 1D DFT is a complex matrix-vector product $y = F_n x$ where x and y are complex vectors, and $F_n = (f_{pq})$ is an $n \times n$ matrix such that $f_{pq} = \omega_n^{pq}$ where $\omega_n = \cos(2\pi/n) - i\sin(2\pi/n) = e^{-2\pi i/n}$ and $i = \sqrt{-1}$. An FFT algorithm exploits structure in F_n to compute the DFT using $O(n \log n)$ floating point operations. See [26] for an excellent description of the numerous FFT algorithms that have been developed over the past 40 years.

Large 1D FFTs, 2D FFTs, as well as higher dimensional FFTs can be computed as a sequence of small, independent, data parallel 1D FFTs with transpose-like operations between them. If $n = n_1 n_2$, then a 1D FFT can be computed using a collection of smaller independent 1D FFTs [2, 13, 26]. Starting with a view of the input vector x as an $n_1 \times n_2$ matrix A, stored in column major order, perform n_1 n_2-point FFTs on the rows of A, scale each element a_{pq} of the resulting matrix by a factor of ω_n^{pq}, and then perform n_2 n_1-point FFTs on the columns of A. The final result vector y is obtained by concatenating the rows of the resulting matrix. Higher dimensional FFTs are computed in a similar fashion. Given an $n_1 \times n_2$ input matrix A, we can compute a 2D FFT by performing n_1 n_2-point 1D FFTs on the rows of A, followed by n_2 n_1-point FFTs on the columns of A.

The issues involved in mapping all FFTs are very similar. Here we choose a 128 $\times$ 128 2D FFT as an example. An Fx 2D FFT program that operates on a stream of m arrays has the following 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 consists of a parallel section, with calls to two task-subroutines inside a loop that iterates m times. Figure 4 shows the task graph for one iteration of the parallel section. Subroutine `rffts` is a data parallel procedure that produces a complex 128 $\times$ 128 array A, distributed by rows, and applies a 1D FFT to each row. Subroutine `cffts` is a data parallel procedure that redistributes A by columns, and applies a 1D FFT to each column. The source of data parallelism within each task-procedure is a parallel loop statement.

Using the framework to map the FFT program

The 2D FFT program consists of two data parallel task-subroutines (`rffts` and `cffts`), and since one array is distributed by rows and the other by columns, the communication step includes a transpose. Here then are the steps we take to map a 128 $\times$ 128 2D FFT onto a 64-processor system, according to the framework described in Section 5:

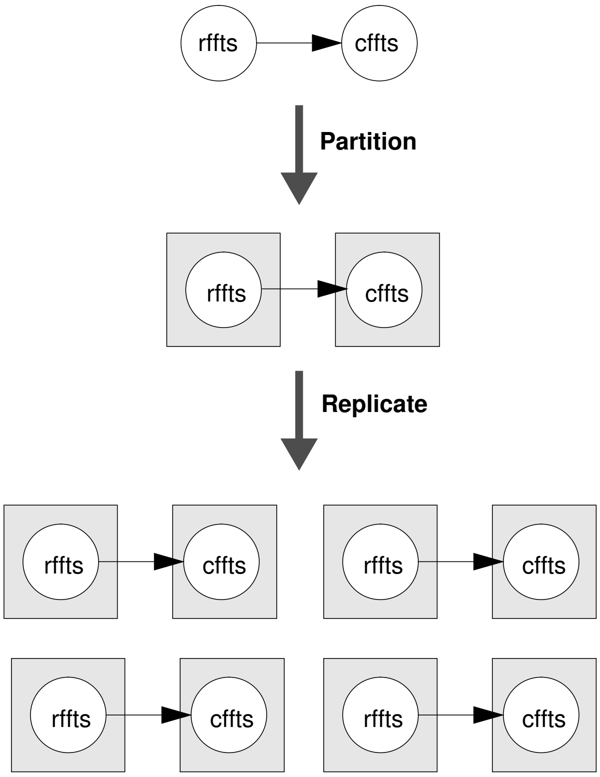

Figure 4: FFT task graph and mapping

Step 1: Clustering. There are only two possible clusterings: assigning the two task-subroutines to the same module or to different modules. For the FFT program, both the task-subroutines can be replicated because the input data sets are independent and because neither of the task-subroutines carries any state. The maximum degree of replication for either task-subroutine is 4, since there are a maximum of four I/O streams (as discussed in the beginning of the section). According to Section 5, we assume a maximum degree of replication, and then make the clustering decision on the basis of performance when a maximum degree of replication is used.

If the two task-subroutines are in the same module, then after maximal replication, each instance of the module executes on 16 processors, and the inter-task communication step uses message passing to redistribute a 128×128 complex array on 16 processors. From performance estimates, one iteration of `cffts` and `rffts` executes in 16.97 ms and 16.60 ms respectively, and the communication step takes 8.10 ms, yielding a total execution time of 41.67 ms. for one iteration on 16 processors.

On the other hand, if the two task-subroutines are in different modules, then after maximal replication, each instance of a module executes on 8 processors, and the communication step is a transfer of a 128×128 complex array between two sets of 8 processors. From the performance estimates, one iteration of `cffts` and `rffts` executes in 34.04 ms and 32.41 ms respectively on 8 processors, and the communication step takes 5.51 ms. The average execution time for one iteration on per 16 processors (taking pipelining into account and assuming perfect load balancing) is ((34.04 + 5.51)*8 + (32.41+5.51)*8)/16 = 38.73 ms.

Since keeping the task-subroutines separately gives a slightly better relative performance estimate (38.73 ms versus 41.67 ms), we put them in different modules. Note that the modules are computationally scalable, and the decision was mainly based on communication estimates.

Step 2: Processor Allocation. Since both modules perform nearly identical computations, each is allocated half (32) of the total number (64) of processors.

Step 3: Replication. Replication is desirable for the FFT because it reduces the total execution time by reducing the cost of the inter-task communication step, but is limited to 4 due to I/O constraints. So we replicate each module 4 times, with each instance of a module running on 8 processors. The resulting mapping for the 128×128 2D FFT program is shown in Figure 4.

Figure 5 shows the performance of the different mappings of the FFT program for different problem sizes. For the 128×128 FFT, the mapping we derived with the framework (2 modules replicated 4 ways) outperforms all others (albeit only slightly the "1 module replicated 4 ways" mapping). Similarly, we are able to derive the best mappings for FFTs of other sizes, but the discussion is omitted for brevity. Notice that the optimal clustering depends on the problem size, but a higher degree of replication always improves performance.

size	one module			two modules		
	pure data parallel	2-way replication	4-way replication	no replication	2-way replication	4-way replication
64×64	105.4	198.1	275.0	219.2	279.3	320.3
128×128	59.12	68.00	75.67	53.80	66.21	76.26
256×256	16.42	17.90	19.05	12.97	15.43	17.73

Figure 5: Measured performance of different size FFTs in datasets/second

6.2 Narrowband tracking radar

The narrowband tracking radar benchmark was developed by researchers at MIT Lincoln Labs to measure the effectiveness of various multicomputers for their radar applications [22]. It is a particularly interesting benchmark for studying task parallelism because of its hard real-time requirements, and because the size of the input data set is limited by physical properties of the radar sensor. The amount of available low-level data parallelism is limited, so additional parallelism must come from higher-level task parallelism.

The radar program inputs data from a single sensor along $c = 4$ independent *channels*. Every 5 milliseconds, for each channel, the program receives $d = 512$ complex vectors of length $r = 10$, one after the other in the form of an $r \times d$ complex matrix A (assuming the column major ordering of Fortran). At a high-level, each input matrix A is processed in the following way: (1) *Corner turn* the $r \times d$ input matrix to form a $d \times r$ matrix. (2) Perform r independent d-point FFTs. (3) Convert the resulting complex $d \times r$ matrix to a real $w \times r$ submatrix, $w = 40$, by replacing each element $a + ib$ in the $w \times r$ submatrix with its scaled magnitude $\sqrt{a^2 + b^2}/d$. (4) Threshold each element a_{jk} of the submatrix using a cutoff that is a function of a_{jk} and the sum of the submatrix elements.

The Fx version of the radar program operating on a stream of m input data sets has the following form:

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

The program consists of a parallel section with calls to two task-subroutines inside a loop that iterates m times. Figure 6 shows the task graph. Task-subroutine `dgen` acquires the data from all 4 channels and sends it to task-subroutine `compute`, a data parallel routine that performs steps (1)–(4) above. The data parallelism is in the form of a parallel loop where each loop iteration operates on a single column of the corner-turned data set.

Using the framework to map the radar program

The radar program has two characteristics that dominate the choice of a mapping. First, the data generation task-subroutine must run on a single processor because it acquires the input data from a sensor that is attached to a particular processor. Second, since there are only 10 columns in the input array after it has been corner turned, and an FFT

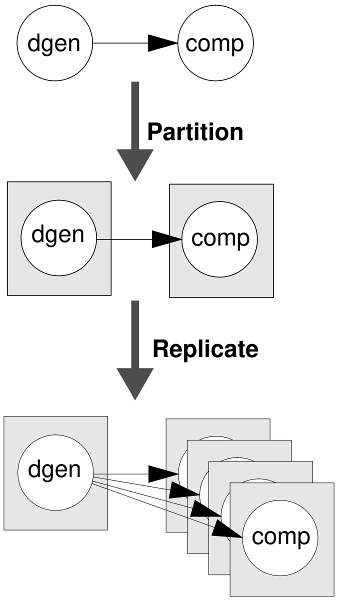

Figure 6: Radar task graph and mapping

replication	1-way	2-way	4-way
datasets/sec	20.2	40.3	80.6

Figure 7: Measured performance of the radar program

computation is done on each column, the amount of data parallelism available to the `compute` task-subroutine is quite small. Following are the steps we take to map the radar program onto 64 processors.

Step 1: Clustering. As with the FFT example, there are two task-subroutines and two possible clusterings. Replication of the `dgen` task-subroutine is not possible because it must run on one processor. However, replication of the `compute` task-subroutine is both possible and desirable: possible because the input data sets are independent, desirable because each instance of the *compute* task-subroutine is able to efficiently use only a small number of processors. If we assign the task-subroutines to the same module, then we eliminate the possibility of replicating the `compute` task-subroutine. So we assign the task-subroutines to different modules, as shown in Figure 6.

Step 2: Processor Allocation. For the radar program, this step is trivial. We assign one processor to the `dgen` module and the remaining 63 processors to the compute module.

Step 3: Replication. Since the input data set is small, and `compute` is not scalable beyond 10 processors, replication is desirable. However, latency constraints in the specification of the application as well as resource constraints in the communication system of the target system limit us to 4 replicated modules, each running on 10 processors, the maximum number that a module can use efficiently in this implementation.

The resulting mapping is shown in Figure 6. Figure 7 gives the measured performance of the Fx radar program when compiled with different degrees of replication. The near linear speedups illustrate the value of replication, for programs like the radar program that operate on small data sets.

6.3 Multibaseline stereo

The multibaseline stereo uses an algorithm developed at Carnegie Mellon that gives greater accuracy in depth through the use of more than two cameras [18]. It is an interesting program for studying task parallelism because it contains significant amounts of both inter-task and intra-task communication[28], and the size of data sets is fixed. Our implementation is adapted from a previous data-parallel implementation written in a specialized image processing language [27].

Input consists of three $m \times n$ images acquired from three horizontally aligned, equally spaced cameras. One image is the *reference image*, the other two are *match images*. For each of 16 disparities, $d = 0, \ldots, 15$, the first match image is shifted by d pixels, the second image is shifted by $2d$ pixels. A *difference image* is formed by computing the sum of squared differences between the corresponding pixels of the reference image and the shifted match images. Next, an *error image* is formed by replacing each pixel in the difference image with the sum of the pixels in a surrounding 13×13 window. A *disparity image* is then formed by finding, for each pixel, the disparity that minimizes error. Finally, the depth of each pixel is displayed as a simple function of its disparity.

The Fx version of the stereo program operating on a stream of s input data sets has the following form:

```
C$        begin parallel
          do i = 1,s
             call dgen(R,M1,M2)
C$           output: R,M1,M2
             do d = 0,15
                call diff(R,M1,M2,DIFF,d)
C$              input: R,M1,M2
C$              output: DIFF
                call error(DIFF,ERR(:,:,d),d)
C$              input: DIFF
C$              output: ERR(:,:,d)
             enddo
             call min(ERR,DISP)
C$           input: ERR
C$           output: DISP
          enddo
C$        end parallel
```

Figure 8 shows the task graph. Task-subroutine `dgen` acquires three 256×240 images from the cameras. Each of the 16 instances of the `diff` task-subroutine is a perfectly data-parallel routine that converts the three input images to a difference image. Each instance of the `error` task-subroutine is a data-parallel routine that sums over a window of pixels in the difference image to produce an error measure for each pixel. Each image is distributed by rows within each task, so a processor needs to exchange rows with its neighbors before the error image can be produced. The outputs from the `error` tasks are passed to `min`, which applies a *min*-reduction to produce the disparity image, and then displays the corresponding depth image.

Using the framework to map the stereo program

The relative performance of different mappings of the stereo program is determined primarily by (1) the cost of the inter-task communication steps that pass images from one task-subroutine to the next (without changing the distribution), and (2) the cost of the intra-task communication steps dominated by the `error` task-subroutines. Following are the steps we take to map the stereo program onto 64 processors:

Step 1: Clustering. For all pairs of communicating task-subroutines in the stereo program, there is no inter-task communication cost if they are mapped to the same module, since the data distributions are the same. However, if we place any two communicating task-subroutines in different modules, there is a significant data movement and communication cost, and there is no gain in terms of replicability or memory requirements. So we decide to place all task-subroutines in a single module, as shown in Figure 8, and use replication to enhance performance.

Step 2: Processor Allocation. The clustering from Step 1 consists of one module, to which we assign all 64 processors.

Step 3: Replication. Memory constraints in our target system limit the degree of replication to four modules, each running on 16 processors. The final mapping is shown in Figure 8. Figure 9 shows the measured performance of the Fx stereo program compiled as 1 (i.e. purely data parallel), 2, and 4 replicated modules. The higher throughput of the 4-way replicated case validates the decision to use replication. However, notice that while a 4-way replication roughly doubles the throughput, it roughly doubles the latency too. Depending on the requirements of a particular application

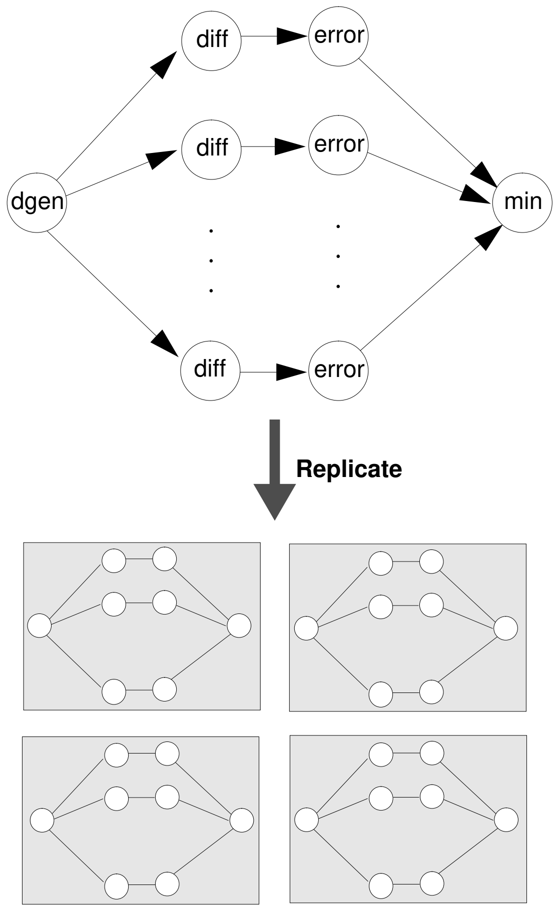

Figure 8: Stereo task graph and mapping

of the stereo program, this may or may not be a reasonable tradeoff. A system striving to minimize latency would potentially arrive at a different mapping.

degree of replication	latency msec/frame	throughput frames/sec
1	112	8.9
2	155	12.9
4	231	17.2

Figure 9: Measured performance of 256×240 stereo

6.4 Summary of example programs

For each example program, we are able to use the framework to derive mappings that exploit a mix of task and data parallelism to achieve better performance than a straightforward data parallel mapping. Each example program exercises different aspects of the framework. In the FFT program, the parallelization overhead is dominated by the communication step between the two FFTs. By clustering into two modules and using replication, the efficiency of communication is maximized. In the narrowband tracking radar program, the main computation routine can effectively use only 10 processors, and improvement is possible by replicating this routine, which in turn is made possible by placing the *datagen* and *compute* routines into separate modules. In the multibaseline stereo program, the cost of communication inside task-subroutines is reduced by replicating the entire computation, which results in each task-subroutine instance executing on fewer processors with better locality.

7 Automatic mapping with real world constraints

In this paper we have demonstrated that a variety of applications can benefit from using a mix of task and data parallelism, and that good mapping decisions can be based on well understood program properties. In related work, we have used this analysis to develop an automatic tool that obtains profile data from program executions, and uses it to automatically generate a good mapping [23].

Our experience is that real applications in signal and image processing often have constraints that cannot be captured easily by a fully automatic system, hence some user involvement in the mapping process is important (see [14] for a discussion of current directives). There are several reasons for this. The mapping is affected by the supporting I/O framework which is hard to model in general. For instance, in the stereo program only specific processors are connected to the camera input, and the number of I/O streams is determined externally. In the radar program, only a single processor receives the input stream, and hence the corresponding module cannot be replicated. Another factor that is difficult to model automatically is the memory hierarchy. While we are able to measure the total memory requirements of a program, it is difficult to model the cache and secondary memory behavior. Also, the effects of congestion in the communication system are extremely difficult to model in a general framework. Finally, some scalability characteristics can be difficult to derive automatically from measurements. For example in the radar program, the program performance scales linearly up to 10 processors, but not at all after that, due to the parallel dimension of the main structure being of size 10.

In summary, automatic tools can be invaluable to a programmer for deciding a mapping, but the current state of the art does not allow a satisfactory fully automatic approach. With the Fx toolchain, the programmers can use an automatic tool to guide mapping, or derive a mapping using a profiling tool and the framework presented in this paper.

8 Concluding remarks

The performance of any parallel program depends on how it is mapped onto a parallel machine. For many programs, a simple strategy that focuses only on task or data parallelism is not adequate, and better results can be obtained using both task and data parallelism.

Finding good mappings of parallel programs is one of the key problems in exploiting task and data parallelism together. This paper identifies the fundamental program properties that define feasible mappings and determine their performance. The communication requirements of a parallel program determine the relative performance of different mappings. The memory requirements determine if a mapping is feasible, since the local memory of each node is finite. Based on the analysis of these properties and machine parameters, we developed a framework for finding good mappings.

We illustrated the mapping process using three realistic parallel programs developed with our compiler. In all cases, the mapping obtained is more efficient than a simple data parallel mapping.

References

[1] AGRAWAL, G., SUSSMAN, A., AND SALTZ, J. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. Tech. Rep. CS-TR-3143 and UMIACS-TR-93-94, University of Maryland, Department of Computer Science and UMIACS, Oct. 1993.

[2] BAILEY, D. H. FFTs in external or hierarchical memory. *The Journal of Supercomputing 4* (1990), 23–35.

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

[4] 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.

[5] CHANDY, M., FOSTER, I., KENNEDY, K., KOELBEL, C., AND TSENG, C. Integrated support for task and data parallelism. *International Journal of Supercomputer Applications 8*, 2 (1994), 80–98.

[6] CHAPMAN, B., MEHROTRA, P., VAN ROSENDALE, J., AND ZIMA, H. A software architecture for multidisciplinary applications: Integrating task and data parallelism. working paper, Feb. 1994.

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

[8] 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), vol. 2, pp. 21–24.

[9] CHOUDHARY, A., NARAHARI, B., NICOL, D., AND SIMHA, R. Optimal processor assignment for a class of pipelined computations. *IEEE Transactions on Parallel and Distributed Systems 5*, 4 (April 94), 439–445.

[10] DINDA, P., GROSS, T., O'HALLARON, D., SEGALL, E., STICHNOTH, J., SUBHLOK, J., WEBB, J., AND YANG, B. The CMU task parallel program suite. Tech. Rep. CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, Mar. 1994.

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

[12] FOX, G. The architecture of problems and portable parallel software systems. Tech. Rep. CRPC-TR91-172, Northeast Parallel Architectures Center, 1991.

[13] GENTLEMAN, W. M., AND SANDE, G. Fast Fourier transforms for fun and profit. In *Proc. AFIPS* (1966), vol. 29, pp. 563–578.

[14] GROSS, T., O'HALLARON, D., AND SUBHLOK, J. Task parallelism in a high performance fortran framework. *IEEE Parallel and Distributed Technology*. To appear.

[15] HIGH PERFORMANCE FORTRAN FORUM. *High Performance Fortran Language Specification, Version 1.0*, May 1993.

[16] 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.

[17] NICOL, D., AND O'HALLARON, D. Efficient algorithms for mapping pipelined and parallel computations. *IEEE Transactions on Computers 40*, 3 (Mar. 1991), 295–306.

[18] OKUTOMI, M., AND KANADE, T. A multiple-baseline stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence 15*, 4 (1993), 353–363.

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

[20] RAMASWAMY, S., AND BANERJEE, P. Processor allocation and scheduling of macro dataflow graphs on distributed memory multicomputers by the PARADIGM compiler. In *Proceedings of the 1993 International Conference on Parallel Processing* (St Charles, IL, August 1993), vol. 2, pp. 134–138.

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

[22] 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.

[23] SUBHLOK, J. Automatic mapping of task and data parallel programs for efficient execution on multicomputers. Tech. Rep. CMU-CS-93-212, School of Computer Science, Carnegie Mellon University, November 1993.

[24] 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), pp. 13–22.

[25] TSENG, C., HIRANANDANI, S., AND KENNEDY, K. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93* (Portland, OR, November 1993), pp. 338–350.

[26] VAN LOAN, C. *Computational Frameworks for the Fast Fourier Transform*. SIAM, Philadelphia, PA, 1992.

[27] WEBB, J. Implementation and performance of fast parallel multi-baseline stereo vision. In *Computer Architectures for Machine Perception* (Dec. 1993), pp. 232–240.

[28] WEBB, J. Latency and bandwidth consideration in parallel robotics image processing˙ In *Supercomputing '93* (Nov. 1993), pp. 230–239.