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

January 1994
CMU-CS-94-106

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, most significantly the memory and communication requirements, and the performance parameters of the target parallel machine. In this paper, we isolate and examine the specific characteristics of programs that determine the performance for different mappings on a parallel machine, and present a framework for obtaining a good mapping. The framework is applicable to applications that process a stream of input, and whose computation structure is fairly static and predictable. We describe three applications that were developed with our compiler: fast Fourier transforms, narrowband tracking radar and multibaseline stereo, examine the tradeoffs between various mappings for them, and show how the framework was used to obtain efficient mappings. The automation of this framework is described in related publications.

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

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 [14]. 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 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, 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 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 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?"

We examine the 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 the program characteristics. The central idea is to find the best use of available memory and communication resources to minimize global inter-task and intra-task communication overheads. We show how the framework was used to map three applications in signal and image processing; Fast Fourier transforms, Narrowband tracking radar and Multibaseline Stereo; and present performance results to show that the mappings chosen were indeed close to optimal.

2 Programming and compiling task parallelism

The Fx compiler supports task and data parallelism. The base language is Fortran 77 augmented with Fortran 90 array syntax, and data layout statements based on Fortran D[15] 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.

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 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[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 module instances, with each instance receiving and sending data sets in round robin fashion. Constraints on replication are discussed in section 5.3. Finally, a subset of the processor array is assigned to each module instance.

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

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

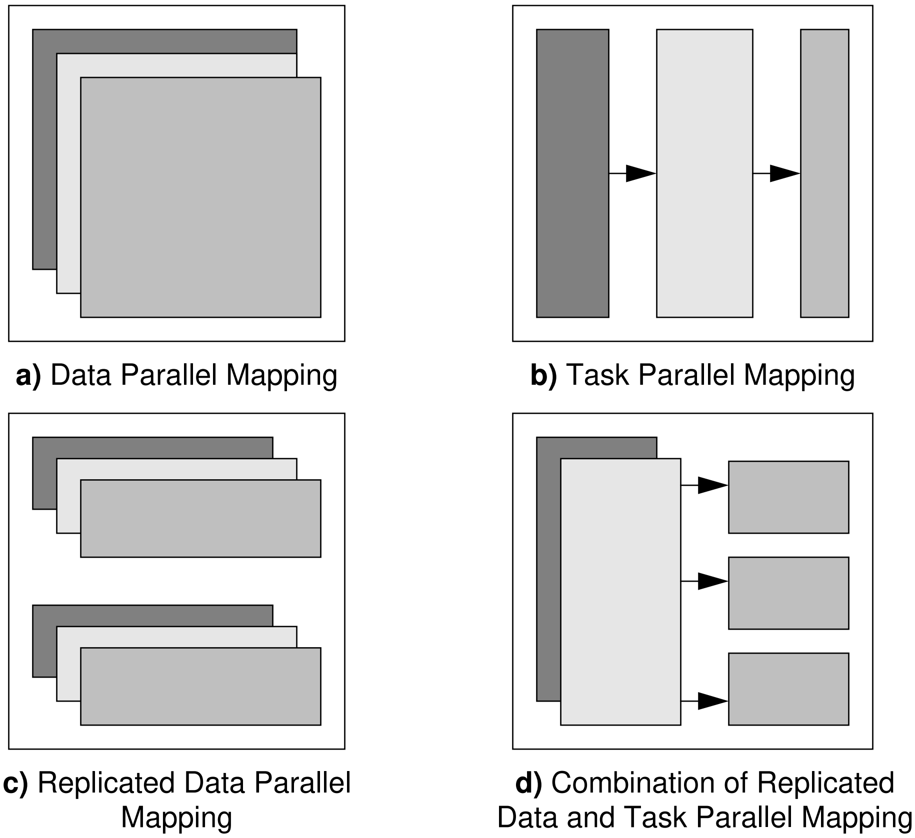

Figure 1: Combinations of Task and Data parallel mappings

```
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
```

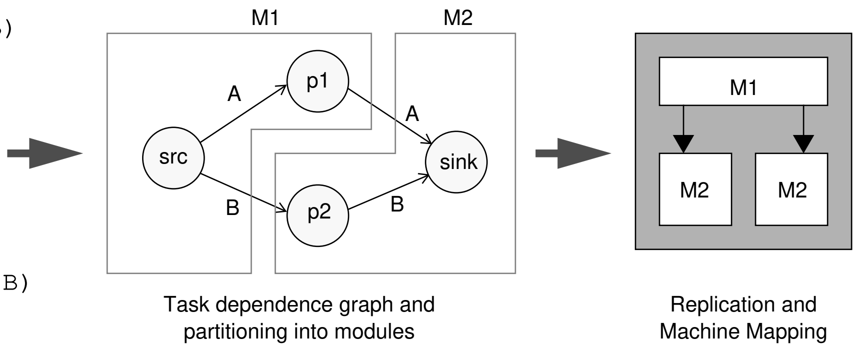

Figure 2: Compilation of task parallelism

the program is mapped onto the machine.

The mapping of the program is controlled by the programmer with directives. In this paper, we develop a framework to guide the programmer in finding a good mapping for a program, and illustrate the procedure with a few applications. A more rigorous description of the framework, along with an automatic mapping tool are discussed in [13].

3 Application domain

The mapping framework discussed in this paper is applicable to programs that process a stream of inputs, and whose behavior is fairly static and predictable. In particular, it is not applicable to programs with dynamic behavior in terms of execution time and size of data structures. We assume that the major program characteristics like execution time and memory requirement can be determined statically, and do not depend on the input data sets. These characteristics can be determined experimentally or analytically. In related research, we have shown that 5 different program executions are enough to determine the main characteristics of a program with sufficient accuracy for the purpose of identifying a good mapping [13]. The mapping procedure assumes knowledge of program and machine characteristics. In many cases, a qualitative idea of the program is sufficient for making a mapping decision, and detailed measurements are not necessary.

Computer vision, image processing and signal processing are examples of application domains where many programs satisfy these requirements, and pure data parallel computing often leads to disappointing results. The example programs chosen are representative of applications in these areas.

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

We now describe these characteristics, and will use them as the basis for our mapping framework discussed 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 processors increases. Plot C represents a purely sequential computation, which cannot use more than 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.

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 communication style used, the number of processors executing task-subroutines, and the data distributions.

Communication style

The style of data exchange between task-subroutines depends on whether they belong to the same or different modules, i.e. whether they are mapped on the same set of processors, or on different sets of processors. When they are in the same module (data parallel style), the data exchange is a local redistribution of data, and for different modules (task parallel style), the data is sent from one set of processors to another set. In our implementation, communication between modules is done systolically over a single channel, which has an extremely low overhead, but limited bandwidth.

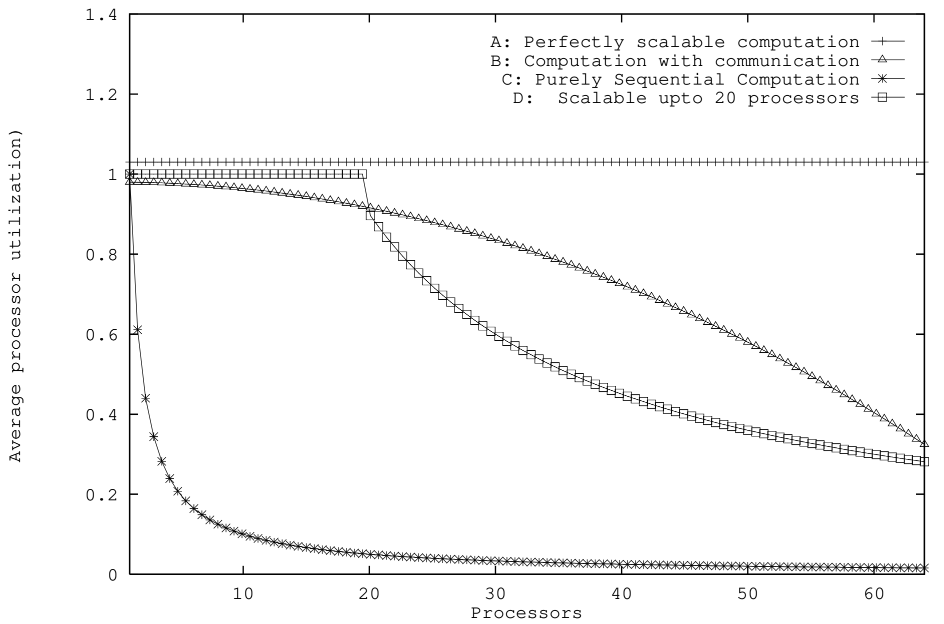

Figure 3: Scalability of computations

Communication inside a module is done using a message passing library, which has a relatively higher overhead, but more parallelism and bandwidth. The nature of the tradeoffs between these styles of communication is illustrated in Figure 4.

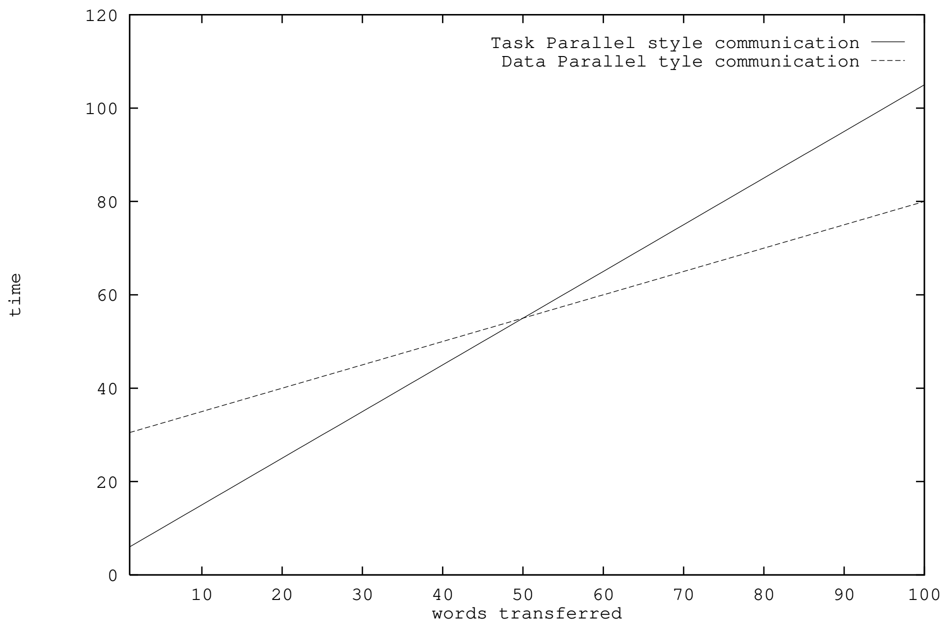

Figure 4: Tradeoff between communication styles

Number of processors

In both the styles discussed above, the cost of communication also depends on the number of processors in the two tasks involved. For communicating a fixed amount of data, the per processor efficiency of communication decreases with increasing number of processors, although the total time for communication may decrease or stay the same.

Data distributions

When the communicating task-subroutines belong to the same module, the cost varies from zero, when the distribution of data is the same in both subroutines, to the cost of a fine grained all-to-all communication, when one distribution is a transpose of the other. For data, as well as task parallel style communication, the granularity, and hence the overhead of communication, is dependent on the data distributions.

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 minimize the execution time. 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 is efficient execution of task-subroutines. This may require that the 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. 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: Partitioning task-subroutines into modules.

Step 2: Allocate processors to modules.

Step 3: Replicate modules into module instances.

We present 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 partitioning decision.

5.1 Partitioning

The first step is to partition 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 the corresponding nodes are assigned to the same or different modules. This problem is very different from partitioning problems addressed in the literature [11, 2]. In our method, the criterion for deciding if a pair of nodes should be assigned to the same or different partitions is most important, and the higher level scheme for partitioning is of less importance.

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 are 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 compare the estimated performance of execution of the computation in the two modules, as well as the communication between them for the two cases. If replication is possible (see next subsection), we assume maximum possible replication in both cases, that is, the modules are assumed to execute on the smallest number of processors they need to execute. This provides an optimistic estimate of the execution time, which is usually sufficiently accurate for partitioning.

The difference in the performance of the merged and the unmerged modules is due to the following factors:

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 has to be transferred between sets of processors. The relative cost is also dependent on the volume of data to be transferred, due to the nature of our implementation, as shown in Figure 4.

At the end of this phase, the task subroutines are partitioned into modules.

5.2 Processor allocation

Based on the execution time estimates of the modules discussed in the last section, the available processors are distributed among them.

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.

The legality and the degree of maximum possible replication is determined by the following constraints:

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

Memory Requirement Every module needs a minimum number of processors to hold the dataset, 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. The extent of latency that can be tolerated is 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 to negate potential benefits. Communication resources also restrict the number of modules that can interface with external I/O agents.

For the purpose of determining a partition, 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 partitioning 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 applications

In this section, we apply the framework developed in the previous section to three realistic task and data parallel applications: fast Fourier transform [14], narrowband tracking radar [12], and multibaseline stereo imaging [17]. We briefly describe each application, show how it is implemented as an Fx program, and show how to use the framework to reason about mapping task parallel programs, and to obtain a good mapping. We present performance results on a 64 processor iWarp array [3] to validate our approach. Each application has unique properties that exercise different parts of the mapping framework. We have found these applications to be invaluable; interested readers can obtain complete and self-contained Fortran 77 and Fx (similar to HPF) implementations from the authors.

All the applications that we describe act on a stream of input in real time, and generate 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. Latency is another important concern for real-time programs. For the purpose of obtaining a mapping, we attempt to optimize throughput, but we specifically address the latency/throughput tradeoffs for the multibaseline stereo application.

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 [16] 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 [1, 8, 16]. 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 128×128 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 5 shows the task graph for one iteration of the parallel section. Subroutine `rffts` is a data parallel procedure that produces a complex 128×128 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.

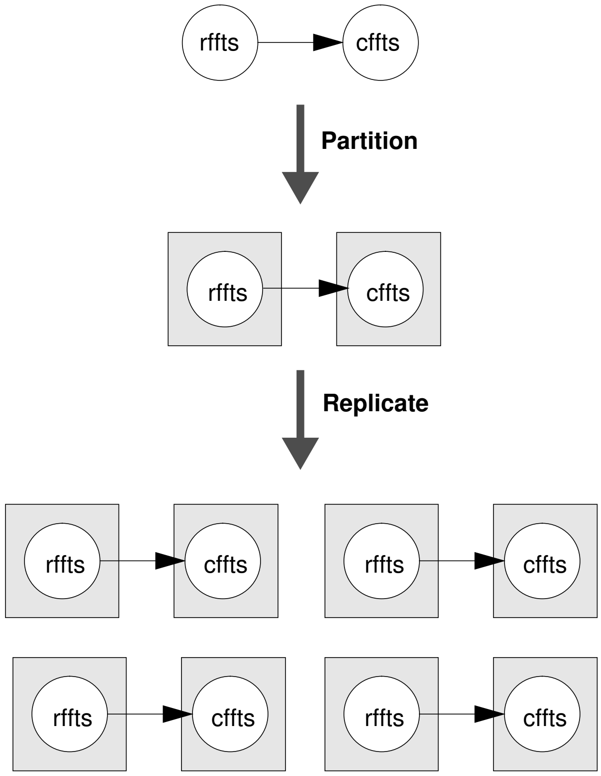

Figure 5: FFT task graph and mapping

Using the framework to map the FFT program

The 2D FFT program consists of two data parallel stages separated by a transpose-like inter-task communication step, that copies an array distributed by rows to an array distributed by columns. The relative performance of different mappings of the program depends entirely on the cost of this communication step. Following are the steps we take to map a 128×128 2D FFT onto a 64-processor system, according to the framework described in Section 5:

Step 1: Partitioning. There are only two possible partitions. According to the framework, in order to compare these partitions, we need to decide whether replication is possible, and if it is possible, whether it is desirable. For the FFT program, replication is possible because the input data sets are independent; it is desirable because it reduces the number of processors involved in the inter-task communication step, and thus reduces its cost.

If the two task-subroutines are in the same module, then after maximal replication (determined by the maximum number of I/O streams discussed in the beginning of this section), each instance of the module executes on 16 processors. The inter-task communication step uses message passing to redistribute a 128×128 complex array on 16 processors. On the other hand, if the two task-subroutines are in different modules, then after maximal replication, each module executes on 8 processors, and the communication step is a systolic transfer of a 128×128 complex array between two sets of 8 processors. On our system, the systolic transfer is (slightly) less expensive than the message passing transfer. Since the computation inside tasks is perfectly scalable, we choose the partition that minimizes the cost of the inter-task communication step. The result of this step is shown in Figure 5.

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

Step 3: Replication. As we noted above, replication is desirable for the FFT because it reduces the cost of the inter-task communication step. However, finite I/O resources in the target system limit the degree of replication to four instances of each module. 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 5. It is important to note that the best mapping depends on the input data size. Figure 6 shows the performance of the FFT program in single precision MFLOPS when the task-subroutines are mapped onto 1 module. Figure 7 shows the performance when the task-subroutines are mapped onto two different modules. For the 64 × 64 FFT, the partition that consists of 2 modules runs significantly faster. On the other hand, partitions that consist of 1 module are better for the 256 × 256 FFT. Note also that the mapping we derived above, with the task subroutines assigned to 2 modules replicated 4 ways, performs slightly better than the alternative of assigning both task subroutines to a single module replicated 4 ways.

size	1-way replication	2-way replication	4-way replication
64 × 64	26	49	68
128 × 128	68	78	87
256 × 256	86	94	96

Figure 6: Measured performance of FFT (in MFLOPS) for the 1-module partition.

size	1-way replication	2-way replication	4-way replication
64 × 64	54	69	79
128 × 128	62	76	88
256 × 256	68	81	93

Figure 7: Measured performance of FFT (in MFLOPS) for the 2-module partition.

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 [12]. 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
```

Like FFT, the program consists of a parallel section with calls to two task-subroutines inside a loop that iterates *m* times. Figure 8 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.

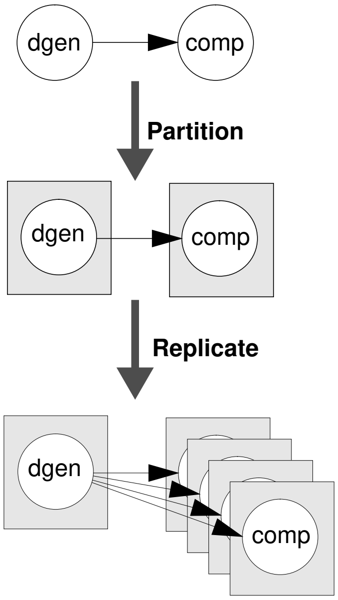

Figure 8: Radar task graph and mapping

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 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: Partitioning. As with the FFT example, there are two possible partitions, and we use knowledge about the feasibility and desirability of replication to help us choose one of these partitions. 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 8.

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, we could maximize the throughput of the radar program by replicating the `compute` module 63 times, running each replicated module on a single processor. However, latency constraints in the specification of the application as well as resource constraints in the communication system of the

replication	1-way	2-way	4-way
time/input(ms)	49.6	24.8	12.4

Figure 9: Performance of radar under different degrees of replication.

target system limit us to 4 replicated modules, each running on 10 nodes, the maximum number that a module can use efficiently in this implementation.

The resulting mapping is shown in Figure 8. Figure 9 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 vision program, which was developed at Carnegie Mellon, uses the information from multiple video cameras to display depth information in real time [17]. It is an interesting program because, unlike the FFT and radar examples, it contains significant amounts of both inter-task and intra-task communication.

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 10 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 it 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.

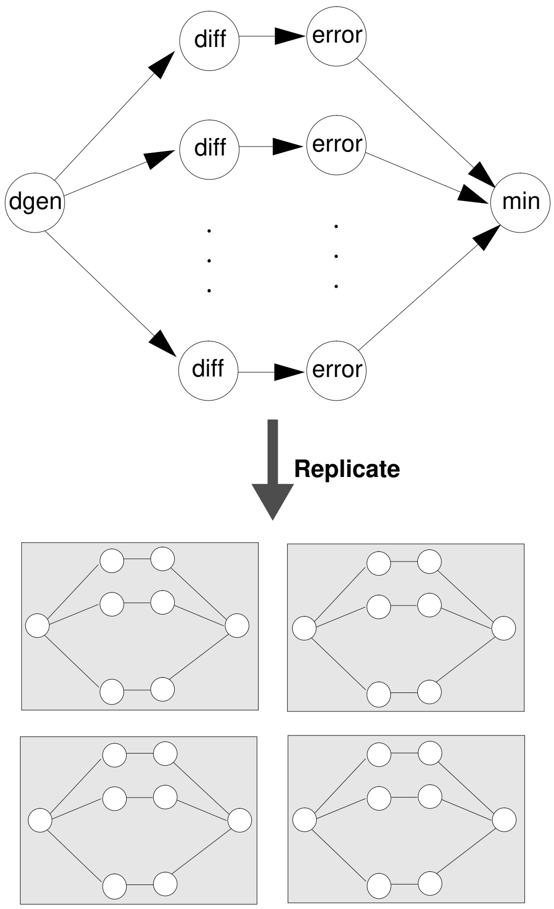

Figure 10: Stereo task graph and mapping

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: Partitioning. For the purpose of mapping, we make the assumption that the I/O system can multiplex the input stream from the cameras onto multiple streams, and hence the data generator task-subroutine can be replicated. Since the input images are independent, the other task-subroutines can be replicated as well. We can increase the efficiency of the communication intensive `error` task-subroutine by partitioning and/or replication. 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 10, and use replication to enhance performance.

Step 2: Processor Allocation. The partition 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 10, and the performance results for a pure data parallel mapping (replication degree 1), and different replicated mappings, are shown in Figure 11.

Latency and throughput tradeoffs in the stereo program

The stereo program captures some potentially important tradeoffs between latency and throughput. Latency is the time required to produce a given frame (an output depth image) from the time the corresponding trio of images is received from the camera. Throughput is the aggregate number of frames produced each second. The mapping procedure described above attempted to optimize the throughput, resulting in a 4 way replicated mapping. Figure 11 shows the measured performance of the Fx stereo program compiled as 1, 2, and 4 replicated modules. Notice that doubling the degree of replication increases throughput by roughly 15%, while roughly doubling the latency. Depending on the requirements of a particular application of the stereo program, this may or may not be a reasonable tradeoff.

degree of replication	LATENCY msec/frame	THROUGHPUT frames/sec
1	161	6.2
2	272	7.3
4	492	8.1

Figure 11: Performance of 256×240 stereo with replication

We now informally discuss the mapping process with the objective of minimizing latency (instead of maximizing throughput). We use a smaller 256×100 stereo program for this discussion [2]. For minimizing latency, the possibility of replication is eliminated. The main tradeoff is between improved performance of individual task-subroutines when there is a larger number of modules, since each task-subroutine is then allocated a smaller number of processors, and the additional cost of the inter-task communication that is introduced.

We discuss the performance of three feasible mappings. Partitioning the 16 *diff* and *error* steps into two different modules leads to the mapping shown in Figure 12 with a total of four modules. This mapping has better latency and throughput than a pure data parallel mapping, but partitioning the *diff* and *error* steps into four modules, leading to a total of six modules, reduces the performance to below that obtained by a pure data parallel mapping. The performance measurements for these partitions are tabulated in Figure 13. The reasons are complex, including the interactions between the different inter-task and intra-task communication steps, which is not fully captured by our simple model.

6.4 Summary of example applications

We summarize how we were able to obtain performance superior to simple data parallel mapping in the three example programs. In the FFT program, the parallelization overhead is dominated by the communication step between the two FFTs. By partitioning into two modules and using replication, a more efficient communication style is used, and a smaller number of processors participate in each communication step, thus reducing the average cost of the communication. In the narrowband tracking radar program, the main computation routine can effectively use only 10 processors, and the improvement was possible by replicating this routine, which in turn was made possible by partitioning the *datagen* and *compute* routines into separate modules. In the multibaseline stereo program, the cost of communication inside task-subroutines was reduced by replicating the entire computation which resulted in each task-subroutine instance executing on fewer processors with better locality. The second implementation of stereo reduces the cost of communication in task-subroutines by mapping different task-subroutines on different subsets of processors, whenever the benefit of improved locality exceeds the cost of additional inter-task communication.

7 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 [15] and Vienna Fortran [4], and for task parallel programs [9, 10]. Recent research shows that a large class of applications contain task and data parallelism [7] and it is important to exploit them in a single compiler

[2] The number of processors in our machine is too small for a meaningful discussion with the full size stereo.

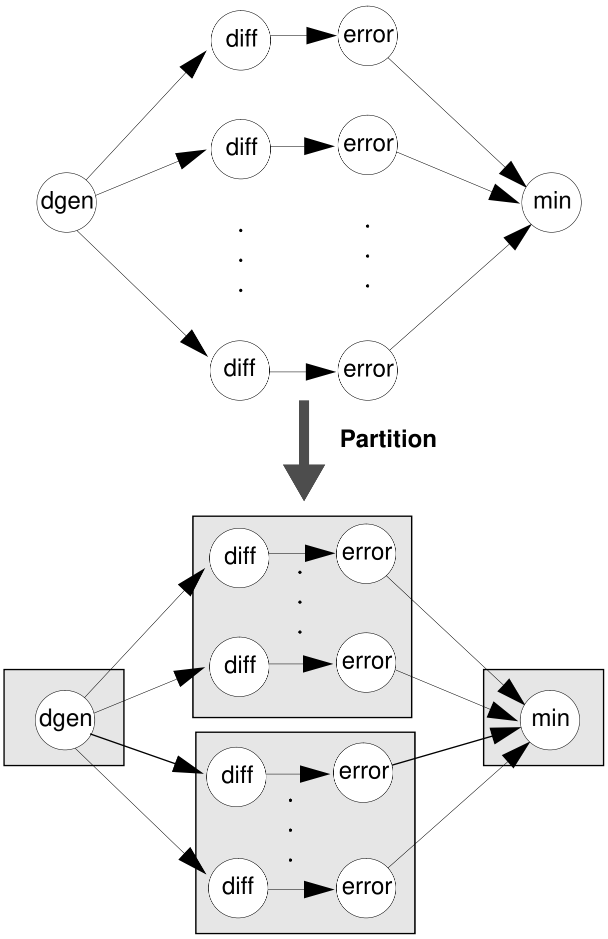

Figure 12: Stereo task graph partitioned into 4 modules

number of modules	LATENCY msec/frame	THROUGHPUT frames/sec
1	112	9.0
4	84	12.0
6	120	8.3

Figure 13: Performance of 256 $\times$ 100 stereo partitioned different ways

framework [5, 6, 14]. There is also a large body of literature on, partitioning, load balancing and scheduling of parallel programs [2, 11].

We have addressed the specific partitioning and mapping issues that arise when task and data parallelism are combined in a parallelizing compiler, based on the characteristics of computations in the program. Our system is part of the Fx compiler that exploits task and data parallelism. Our approach is different from Fortran M [6] 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. An alternate approach, taken in Jade [9] 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 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.

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 have developed a framework for finding good mappings.

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

9 Acknowledgements

The entire Fx compiler group at Carnegie Mellon contributed to this research. Jim Stichnoth and Bwolen Yang developed the compiler that was used to compile data parallel modules. Susan Hinrichs developed the Programmed Communication System that we used for inter-task communication.

References

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

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

[3] BORKAR, S., COHN, R., COX, G., GROSS, T., KUNG, H. T., LAM, M., MOORE, M. L. B., MOORE, W., PETERSON, C., SUSMAN, J., SUTTON, J., URBANSKI, J., AND WEBB, J. Supporting systolic and memory communication in iWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, WA, May 1990), pp. 70–81.

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

[5] 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).

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

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

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

[9] 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).

[10] 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.

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

[12] 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.

[13] 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.

[14] 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).

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

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

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