IEEE Parallel & Distributed Technology, Volume 2, Number 3, Fall, 1994, pp. 16-26

Task Parallelism in a High Performance Fortran Framework

T. Gross, D. O'Hallaron, and J. Subhlok
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

High Performance Fortran (HPF) has emerged as a standard dialect of Fortran for data parallel computing. However, for a wide variety of applications, both task and data parallelism must be exploited to achieve the best possible performance on a multicomputer. We present the design and implementation of a Fortran compiler that integrates task and data parallelism in an HPF framework. A small set of simple directives allow users to express task parallel programs in a variety of domains. The user identifies opportunities for task parallelism, and the compiler handles task creation and management, as well as communication between tasks. Since a unified compiler handles both task parallelism and data parallelism, existing data parallel programs and libraries can serve as the building blocks for constructing larger task parallel programs. This paper concludes with a description of several parallel application kernels that were developed with the compiler. The examples demonstrate that exploiting data and task parallelism in a single framework is the key to achieving good performance for a variety of applications.

1. Introduction

Compilation of programs for parallel computers has received considerable attention for many years. Several parallelizing compilers have been developed for data parallel programs, including Fortran D [26] and Vienna Fortran [9]. High Performance Fortran [15] (HPF) has emerged as a standard dialect of Fortran for data parallel computing. The core of HPF contains a set of extensions to describe data mappings and parallel loops. These allow programmers to write and compile data parallel programs for a variety of architectures. However, in its current form, HPF does not address *task parallelism* or *heterogeneous* computing adequately. Applications that require different processor nodes to execute different programs, possibly on different data sets, cannot be programmed effectively in HPF. There is growing interest in the idea of exploiting both task and data parallelism [1, 7, 8, 10, 11, 12, 14, 25]. There are a number of practical reasons for this interest:

Limited scalability: Many applications, especially in the domains of image and signal processing, do not scale well when using data parallelism, because data set sizes are limited by physical constraints, or because they have a high communication overhead. For example, in multibaseline stereo [27], the main data set is an image whose size is determined by the camera interface. Task parallelism makes it possible to execute individual computations on a subset of nodes and thus improves performance, despite limited scalability.

Real-time requirements: Many real-time applications (e.g. in robot control) have strict latency and throughput requirements. Task parallelism allows the programmer to partition resources (including processor nodes) explicitly among the application modules to meet such requirements. By supporting *both* task and data parallelism in a single framework, the user can tailor the mapping of an application to a particular performance goal.

Multidisciplinary applications: Task parallelism can be used to effectively manage heterogeneity in applications and execution environments. There is an increased interest in parallel multidisciplinary applications where different

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

modules represent different scientific disciplines and may be implemented for different parallel machines. For example, the airshed model [16, 17] represents a “grand challenge” application that characterizes the formation of air pollution as the interaction between wind and reactions among various chemical species. It is natural to model such interactions using task parallelism; e.g. one module (or task) models the effect of the wind, and a different module models the chemical reactions. Further, the use of task parallelism is necessary if different modules are designed to execute on different types of parallel or sequential machines.

There are many ways in which task and data parallelism can be supported together in a programming environment. A fundamental design decision is whether the programmer has to write programs with explicit communication, or if the responsibility of communication generation is delegated to the compiler. One of the benefits provided by data parallel languages like HPF is that they liberate the programmer from dealing with the details of communication, which is a cumbersome and error prone task. If task parallelism is to find acceptance, writing task parallel programs must be no harder than writing data parallel programs, and therefore, in our design, all communication operations are generated by the compiler. The user writes programs for a common data space, and the compiler maps the data objects to the (possibly disjoint) address spaces of the nodes of the parallel system. This division of responsibility also allows communication optimizations by the compiler. Other fundamental design decisions include whether task management should be static or dynamic, and strategies for processor allocation and load balancing.

We have designed and implemented task parallelism as directives in a data parallel language based on HPF. This prototype compiler is called Fx [25, 22].[1] Our objectives are to develop a system that produces efficient code, and to use this system to develop applications that need task and data parallelism. The current targets for this compiler are an iWarp parallel machine, networks of workstations running PVM, and the Cray T3D. The compiler has been used to develop a variety of task and data parallel applications, including synthetic aperture radar, narrowband tracking radar, and multibaseline stereo [13, 24].

There are obvious practical advantages of extending HPF for task parallelism instead of inventing a new language. Existing sequential and data parallel libraries can be used, it is easier to convert existing programs to task and data parallel programs, and it is easier to gain user acceptance. Finally, it is important to be able to compile task and data parallel programs efficiently using existing compiler technology. In particular, we allow several directives to help the compiler in generating efficient code, even though some directives may become obsolete as more sophisticated compilers become available.

The paper is organized as follows. Section 2 provides the fundamental motivation for exploiting task and data parallelism and introduces an example program, which is re-used throughout the paper to illustrate various concepts. Section 3 describes the user model that Fx presents for writing task and data parallel programs. Section 4 discusses the main issues involved in compiling such programs and describes how task parallelism and data parallelism coexist in one framework. Section 5 reports experience with applications written for the Fx compiler; this section demonstrates the performance benefits that can be obtained by finding the right balance between task and data parallelism.

2. Requirements for efficient parallelization

Many applications must exploit both task and data parallelism for efficient execution on massively parallel programs. Consider the following example application kernel (called FFT-Hist) from signal and image processing. Input is a sequence of m 512×512 complex arrays from a sensor (e.g., a camera). For each of the m input arrays, we perform a 2D fast Fourier transform (FFT), followed by some global statistical analysis of the result, including constructing a histogram. The 2D FFT consists of a 1D FFT on each column of the array, followed by a 1D FFT on each row. The main loop nest in FFT-Hist is shown in Figure 1.

For each iteration of the loop, the `colffts` function inputs the array `A` and performs 1D FFTs on the columns, the `rowffts` function performs 1D FFTs on the rows, and the `hist` function analyzes and outputs the result. This is an

[1]There are two explanations for this name. On one hand, the "x" emphasizes that the language and directives may still undergo further development, and the "F" emphasizes how irrelevant details of the base language (Fortran) are. On the other hand, efficient translation of programs for parallel machines often brings to mind the use of special effects.

Figure 2: Speedup curves for the functions in FFT-Hist.

Given that only two out of the three functions scale well, how do we go about parallelizing the loop nest in Figure 1? One approach is a purely data parallel mapping: use all of the nodes to execute `colffts`, then use all of the nodes to execute `rowffts`, then use all of the nodes to execute `hist`, and so on. As the number of nodes increases, this purely data parallel approach works well for the `colffts` and `rowffts` functions but makes inefficient use of the nodes during the `hist` routine because `hist` does not scale well.

To achieve good efficiency for functions like the `hist` function, we must allocate a small number of nodes to it. So for large parallel systems, how do we use up the remaining nodes? The answer is to exploit a mix of task and data parallelism.

3. User model for task and data parallelism

The input language for Fx is based on HPF: The array statments of Fortran 90 augmented with data layout statements and a FORALL-like parallel loop construct [29]. These constructs are described briefly in Section 3.1.

In the Fx model, a *task* corresponds to the execution of a call to a *task-subroutine*. A task-subroutine is a data parallel subroutine, with well-defined side-effects, contained inside a special code section in the main program called a *parallel section*. The only allowable side-effect of calling a task-subroutine is that the values of its actual parameters might change. For each lexical call to a task-subroutine, the programmer provides (1) hints that indicate if an actual parameter is read and/or modified by the task subroutine, and (2) directives that control the mapping of the task-subroutines onto nodes. These hints and directives are described later in Section 3.2.

The execution model for an Fx program is as follows: The program begins execution as a single data parallel task running on all nodes. When the flow of control reaches a parallel section, the tasks specified by calls to task-subroutines inside the parallel sections are executed subject to data dependence constraints, i.e., each task waits for its input, executes, sends its output, and terminates. Parallelism is obtained by executing different tasks on different sets of nodes. When all tasks have terminated, the execution of the parallel section is over, and the program continues execution as a single data parallel task.

Figure 3 depicts some possible executions of FFT-Hist for $m = 4$ iterations on a parallel system. Details of the organization of the parallel system do not matter at this time. For each node, this figure indicates what function of FFT-Hist is executed on this node at a given time. In Figure 3(a), the main program starts on all of the nodes. Once inside the parallel section, the task-subroutines execute one after the other; each task-subroutine runs on all of the nodes. After 4 iterations of the loop, the main program resumes executing on all the nodes. Another possibility is shown in Figure 3(b), where each task-subroutine runs on a disjoint set of nodes, and thus the computation is pipelined. Notice that the `hist` function takes about the same time as in the mapping of Figure 3(a) – using more nodes did not shorten execution time in Figure 3(b). Yet another option is depicted in Figure 3(c).

Since the data relationship of the calling program to the task-subroutines is well defined, the compiler can map the tasks on different sets of nodes, and generate communication to maintain data consistency. For our application domains, the runtime behavior of tasks can be accurately predicted before execution, so issues like load balancing and task migration are currently not of concern to our compiler. Load balancing can be influenced by the user's choice of data layout, using the HPF layout directives.

The basic idea governing the role of directives is that the results obtained from parallel execution must be consistent with those obtained from sequential execution. The main characteristics of the user model can then be summarized as follows: (1) There are no new language constructs, only compiler directives in the form of comments. (2) There is a common name space for shared data. (3) Tasks are represented as calls to data parallel subroutines with well-defined side-effects. (4) Communication between tasks is generated and managed by the compiler. (5) Sequential consistency, determinism, and freedom from deadlock are guaranteed by the compiler.

The rest of this section discusses these ideas in more detail. Section 3.1 gives a brief overview of the data parallel constructs, and Section 3.2 describes the directives available to users.

3.1. Data parallel constructs

Data parallelism is expressed with array statements (as in HPF) and a FORALL-like parallel loop called the PDO [29]. Fx supports BLOCK, CYCLIC, and BLOCK-CYCLIC distributions in an arbitrary number of array dimensions. Consider the following example:

```
c$      template t(n)
c$      align A(i,j) with t(i)
c$      align B(i,j) with t(j)
```

Figure 3: Execution of FFT-Hist on a parallel system

```
c$      distribute t(CYCLIC)

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

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

3.2. Task parallel directives

We have not introduced any new language features and rely entirely on compiler directives for expressing task parallelism. To simplify the implementation, the current version of Fx relies on the user to identify the side effects of the task-subroutines and to specify them. Directives are also used to guide the compiler in making performance related decisions like program mapping. In this section, we describe the directives and hints that are used to express task parallelism and illustrate their use for the FFT-Hist example.

Parallel sections

Calls to task-subroutines are permitted only in special code regions called *parallel sections*, denoted by a `begin parallel/end parallel` pair. For example, the parallel section for the FFT-Hist example has the following form:

```
c$      begin parallel
        do i = 1,m
           call colffts(A)
c$         input/output and mapping directives
           call rowffts(A)
c$         input/output and mapping directives
           call hist(A)
c$         input/output and mapping directives
        enddo
c$      end parallel
```

The code inside a parallel section can only contain loops and subroutine calls. These restrictions are necessary to make it possible to manage shared data and shared resources (including nodes) efficiently at compile time.

A parallel section corresponds to a mapping of task-subroutines to nodes. The corresponding mapping outside the parallel section is a simple data parallel mapping, where every routine is mapped to all nodes. The current implementation does not allow nesting of parallel sections.

Input/output directives

The user includes `input` and `output` hints to define the side-effects of a task-subroutine, i.e., the data space that the subroutine accesses and modifies. Every variable whose value at the call site may potentially be used by the called subroutine must be added to the input parameter list of the task-subroutine. Similarly, every variable whose value may be modified by the called subroutine must be included in the output parameter list. A variable in the input or output parameter list can be a scalar, an array, or an array section. An array section must be a legal Fortran 90 array section, with the additional restriction that all the bounds and step sizes must be constant.

For example, the input and output directives for the call to `rowffts` have the form:

```
            call rowffts(A)
c$          input (A), output (A)
c$          mapping directives
```

This tells the compiler the subroutine `rowffts` can potentially use values of, and write to the parameter array `A`. As another example, the input and output directives for the the call to `colffts` has the form:

```
            call colffts(A)
c$          output (A)
c$          mapping directives
```

This tells the compiler that subroutine `colffts` does not use the value of any parameter that is passed but can potentially write to array `A` (which is set to values read from a sensor by `colffts`).

Mapping directives

Exploiting task and data parallelism together opens a variety of ways to map a computation onto a parallel machine. In the Fx model, we characterize mappings in terms of three attributes: *clustering*, *degree of replication*, and *node allocation*.

A *clustering* is an assignment of task-subroutines to *modules*. At run time, each task-subroutine in a module runs on the same set of nodes, and each module runs on a unique set of nodes. For example, Figure 4(a)–(c) shows three possible clusterings of FFT-Hist. Figure 4(a) shows the familiar *data parallel mapping* where all task-subroutines are assigned to the same module; this corresponds to the schedule in Figure 3(a). Figure 4(b) shows a *purely task parallel mapping* where each task-subroutine is assigned to a different module; this corresponds to the schedule in Figure 3(b). Figure 4(c) shows a mapping that is a mix of both.

If the data sets in the input sequence of a module are independent, and the module carries no internal state, then that module can be *replicated*. Each copy of the module is called a *module instance*. If the module is replicated into *k* instances, then we say that the mapping uses *k-way* replication, or equivalently, that the *degree of replication* for that module is *k*. Module instances execute the calls to the corresponding subroutines in a round robin order such that each instance executes only *1/k*th of the total number of calls (except for boundary conditions). For example, Figures 4(d)–(e) show mappings with 2-way replication for all modules; one replicated instance executes the even-numbered iterations, and the other replicated instance executes the odd-numbered iterations . In Figure 4(f), the first module is not replicated (i.e., there is only one instance), and the second module is replicated into 4 instances; this corresponds to the schedule in Figure 3(c).

Finally, there is an assignment of nodes to module instances. This attribute is approximated graphically in Figure 4 by the relative sizes of the rectangles. For example, in Figure 4(c), each module instance is assigned half of the nodes. In Figure 4(f), the single instance of the first module is assigned 24 of the available 64 nodes, and each instance of the second module is assigned 10 nodes each.

Often the programmer has a good idea of how a computation should be mapped but does not want to deal with the low level details of the mapping. To allow a programmer to pass this information to the compiler, we include mapping directives. By their very nature, the effect of such mapping directives is machine specific (the directives are not). For example, a user may want to indicate that some sets of tasks be mapped to physically adjacent nodes. The number of

Figure 4: Combinations of task and data parallel mappings.

nodes that are adjacent depends on the architecture of the target machine (4 for a 2D-torus, 6 for a 3D-torus, etc.), but in our experience, such machine-specific hints can improve the performance dramatically. Nevertheless, these directives have no semantic meaning; if ignored by the compiler, performance may suffer but correctness is maintained.

Fx includes the `processor` and `origin` directives to describe the clustering of task-subroutines into modules, the allocation of nodes to modules, and the replication of modules. The `processor` directive states how many nodes should be assigned to a task-subroutine. The `origin` directive states the location(s) of the task-subroutine in the parallel system. In the current implementation, only rectangular subarrays can be assigned to task-subroutines, and the parallel system is assumed to be organized as a two dimensional space, with node (0,0) at the top left of the system. Hence `processor` and `origin` directives contain pairs of numbers referring to the size and location of a rectangular subarray of nodes, respectively. For example, to map FFT-Hist as shown in Figure 4(f) onto an 8×8 array of nodes, we use the following mapping directives:

```
c$      begin parallel
        do i = 1,m
           call colffts(A)
c$         output (A)
c$         processor (8,3)
c$         origin (0,0)
           call rowffts(A)
c$         input (A), output (A)
c$         processor (2,5)
c$         origin (0,3), (2,3), (4,3), (6,3)
           call hist(A)
c$         input (A)
c$         processor (2,5)
c$         origin (0,3), (2,3), (4,3), (6,3)
        enddo
c$      end parallel
```

These directives instruct the compiler that `colffts` should be allocated an 8×3 module of nodes, with the top-left corner of the module at node (0,0). Task-subroutines `rowffts` and `hist` are to be placed on the same 2×5 module, replicated 4 ways, with the top-left corner of the 4 module instances starting at nodes (0,3), (2,3), (4,3), and (6,3), respectively. The replicated instances of the `rowffts-hist` module are called in round-robin order. So, instance 0 gets the first data set, instance 1 gets the second data set, and so on.

The current implementation of Fx only supports homogeneous parallel system, for which the size and location of a subarray is sufficient information to map a task-subroutine. In a heterogeneous environment with different machines, additional information is needed.

4. Compiling task parallel programs

The compiler must perform a set of steps to support task parallelism: (1) Identify the task structure of the program and determine the placement of task-subroutines. This step determines the mapping of the application on the parallel system. (2) Determine the communication links between the task-subroutines and identify the data to be transferred. (3) Generate and schedule inter-task communication. (4) Generate a final program along with variable declarations to manage the shared address space.

The different tasks in the program are obtained by examining the statements in a parallel section. The placement of the tasks in the parallel system is obtained from the mapping directives, i.e. the `processor` and `origin` directives. The dependences between tasks are identified by data flow analysis over array sections using the information in the input and output directives supplied by the user. The task dependence edges are also the communication edges, and the actual communication is generated using the task-communication graph and the data distribution information that is present in the form of alignment and distribution directives inside task-subroutines. Declaration and distribution of array variables, and node assignments determine the amount of memory allocated for array variables on individual nodes.

4.1. Mapping criterion

The mapping of the tasks of a parallel program to the processor nodes is an important determinant of performance. The directives that control mapping may be provided by the user, or generated by an automatic mapping tool. The situation is analogous to the data layout directives in HPF. The mapping process is discussed in more detail in [24], and an automatic mapping tool is discussed in [23]. Here we briefly discuss the basic mapping criterion, which is the same whether the mapping is done by hand or by an automatic tool.

In our experience, the following three dimensions have the biggest impact on the quality of a mapping:

Scalability: When a computation or a subroutine is not scalable, better node efficiency is achieved by using a smaller number of nodes for each computation instance.

Memory requirements: The minimum number of nodes needed for a computation is bounded by memory requirements. This is an important consideration that is often overlooked in the mapping literature.

Inter-task communication: The nature and cost of inter-task communication depends on the mapping. If two tasks are placed in the same module, the cost of the inter-task communication is different than if they are placed in different modules.

The major steps in generating a mapping are:(1) Cluster task-subroutines into modules. (2) Allocate nodes to modules. (3) Replicate modules into module instances.

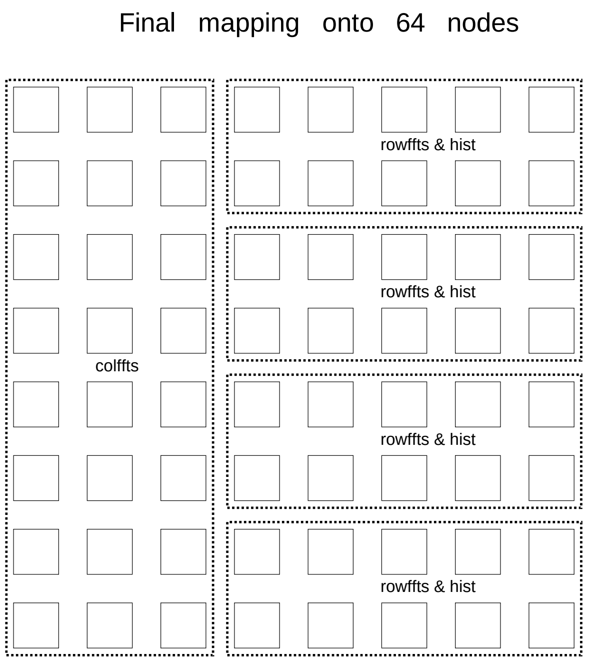

Figure 5: Mapping steps and the final mapping of the example program

In summary, a combined task and data parallel mapping is often needed to achieve the best performance, and the choice is based on measurable program properties. For example, Figure 6 shows the performance of the mapping in Figure 5 on a 64-node iWarp system, relative to a direct data parallel mapping. The mapping in Figure 5, which consists of a mix of task and data parallelism, outperforms the straightforward data parallel mapping by a factor of two.

5. Experience with Fx applications

We used Fx for problems in a variety of domains: medical image processing, synthetic aperture radar, narrowband tracking radar, computer vision, and airshed modelling [13]. This section describes our experience with a subset of these applications and kernels: 2D fast Fourier transform, narrowband tracking radar, and multibaseline stereo

Program Mapping	Speedup over data parallel mapping
Data Parallel (Fig. 4(a))	1
Task Parallel (Fig. 4(b))	1.43
Mixed Mapping (Fig. 5)	1.95

Figure 6: Speedup for different mappings of the FFT-Hist example

imaging [13, 24]. The programs are compiled for a 64-node iWarp system [3, 4]. Since the details of the target machine are not relevant in this context, we present the results as speedup over a purely data parallel implementation. In all the examples, significant performance benefits are realized by compiling the programs with a mix of task and data parallelism.

5.1. Fast Fourier transform

The FFT-Hist example from the previous sections consists of a 2D FFT (task-subroutines `colffts` and `rowffts`), followed by a histogram. The 2D FFT is an interesting application in its own right; even though it shares much of the same code with the FFT-Hist example, it scales differently, and thus its best mapping is quite different.

Figure 7 shows the speedups for different mappings of the FFT program relative to a simple data parallel mapping, for different problem sizes. The numbers are relative only to numbers in the same row and are not comparable across rows. Notice that the optimal clustering depends on the problem size, but a higher degree of replication always improves performance. For example, for the 128×128 2D FFT (a size frequently encountered) a 4-way replication of two modules, as shown in Figure 8, minimizes execution time. This mapping differs from the mapping of FFT-Hist in Figure 5.

size	one module			two modules		
	pure data parallel	2-way replication	4-way replication	no replication	2-way replication	4-way replication
64×64	1	1.88	2.61	2.08	2.65	3.04
128×128	1	1.15	1.28	.91	1.12	1.29
256×256	1	1.09	1.16	.79	.94	1.08

Figure 7: Speedup of 2D FFT relative to a data parallel mapping.

As the problem size increases, the pure data parallel mapping begins to perform better relative to the mixed mappings. The reason is due to differences in the scalability of inter-task communication; in our implementation, communication between task-subroutines in the same module scales better than communication between task-subroutines in different modules. The crucial point here is that the best mapping depends on the input size; no single approach works best in all cases.

5.2. 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 [21]. 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.

Figure 8: FFT task graph and mapping

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 get(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 9 shows the task graph. Task-subroutine `get` acquires the data from all 4 channels and sends it to task-subroutine `compute`, a data parallel routine that performs steps (1)–(4) above. We will assume for purposes of discussion that the `get` task-subroutine must run on one node, and that it must be assigned to the node that is connected to the radar sensor. The data parallelism in the `compute` task-subroutine is in the form of a parallel loop where each loop iteration operates on a single column of the corner-turned data set. Since there are only $r = 10$ of these columns for each of the 4 channels, the amount of loop-level parallelism is quite small.

Since the `get` task-subroutine must run on exactly one node, we can only replicate the `compute` task-subroutine if the two task-subroutines are clustered into different modules. The `compute` task-subroutine can use at most 10 nodes

Figure 9: Radar task graph and mapping

replication	1-way	2-way	4-way
speedup	1	2.000	3.996

Figure 10: Speedup of radar for different degrees of replication.

efficiently, so we want to use up nodes by using replication. A mapping of the program that uses 4-way replication of the `compute` task-subroutine is shown in Figure 9.

Figure 10 gives the measured performance of the Fx radar program when compiled with different degrees of replication. The linear speedups illustrate the value of replication for programs like the radar program that operate on small data sets.

5.3. Multibaseline stereo

The multibaseline stereo example is based on an algorithm developed at Carnegie Mellon for depth perception by using 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 11 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 node 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.

A mapping of the stereo program that uses 4-way replication is shown in Figure 11. Figure 12 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, while a 4-way replication roughly doubles the throughput, it roughly doubles the latency too. Depending on the requirements of a particular application 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.

6. Comparison with related work

The approach that we have taken towards task parallelism can be summarized by the following key features:

- Task parallelism is integrated with a data parallel compiler, and data parallel subroutines are units for task parallelism.
- Task parallelism is expressed with high level directives, and communication and task management is done by the compiler.

Task parallelism that can be expressed in our system is constrained in two significant ways. First, communication between task-subroutines is permitted only at procedure boundaries (through procedure arguments). Since we are using data parallelism with its own compiler-generated communication inside subroutines, there is some justification that explicit communication between task-subroutines is less important. This constraint considerably simplifies the programming model and the compiler. Second, the mapping of tasks to nodes is fixed at compile time. This makes it easier to generate efficient parallel programs with low execution overheads, but makes the method not suitable for dynamic computations.

Coordination languages like Linda [5, 6] and Fortran M [14] provide a communication interface to build task parallel programs, with facilities for more general inter-task communication. In contrast, Fx task parallelism is more

Figure 11: Stereo task graph and mapping

degree of replication	speedup
1	1
2	1.45
4	1.93

Figure 12: Speedup of stereo for different degrees of replication

restricted but is closely integrated with a data parallel compiler, and communication is exclusively generated by the compiler.

Jade [19] and PYRROS [30] capture all parallelism as fine grain task parallelism; these systems create and schedule tasks dynamically. A new language is developed in Jade, and fine grain directives are required in PYRROS. While these systems can support a richer variety of parallelism, particularly dynamic programs, writing programs is more cumbersome because a fine grain control of parallelism by the programmer is required, or because they do not use a standard data parallel layer like HPF, which we found invaluable for ease of development and user acceptance.

HPF [15] can be used to develop task parallel MIMD programs using *INDEPENDENT* parallel loops, but no support is available for expressing data transfers between tasks. Chapman et. al. [8] propose a similiar, but more general and dynamic approach to task parallelism than ours. Fx emphasizes simplicity to obtain efficient code through compilation; it will be interesting to compare the performance once results from an implementation are reported.

The node mapping problem introduced in the paper has been addressed in more detail in related publications [23, 24]. This problem is quite different from the many partitioning problems addressed in the literature (e.g., [20, 2, 11]) due to one or more of the following reasons: (1) Task-subroutines are to be mapped to groups of nodes, not individual nodes. (2) The computation and communication costs are functions of the number of nodes, not constants. (3) The objective is to maximize throughput for a sequence of inputs, not to minimize execution time of a fixed set of tasks.

7. Conclusions

Both task and data parallelism are important for practical applications and are necessary to make the best use of a parallel system. We demonstrate that a set of simple directives is sufficient to capture task parallelism for representative applications in computer vision, signal processing, and multidisciplinary scientific computing. Without task parallelism, it may be impossible to utilize a large number of nodes efficiently. Applications in these domains often exhibit only a limited amount of data parallelism due to the fixed size of their input sets, are subject to real-world latency constraints, or are structured so that individual components scale differently.

The Fx compiler is a prototype system that integrates both data and task parallelism, and our experience demonstrates that task parallelism can be supported effectively in an HPF framework. The current design reflects our desire to obtain a working system that can serve as a basis for further experimentation with the limited resources available. The Fx compiler represents approximately a 10 person-year effort (this includes dealing with task as well as data parallelism). The design contains some limitations: task parallelism is subject to several constraints, and the programmer has limited control over execution and communication. However the design and compiler have proven adequate for interesting applications.

We take the approach that the user provides a high level specification of task parallelism via directives, and the compiler manages the execution of tasks as well as communication between them. This extends one of the attractions of HPF in that it frees the user from dealing with the details of communication in the parallel program. Furthermore, this approach provides the compiler opportunities for optimizing communication and mapping of the program. Our prototype demonstrates the benefits of task parallelism; programs with both styles of parallelism exhibit improved performance over data or task parallelism alone. Fx presents a simple approach to task parallelism that considerably enhances the power of a data parallel language like HPF.

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] BOKHARI, S. H. Partitioning problems in parallel, pipelined and distributed computing. *IEEE Transactions on Computers 37,* 1 (Jan. 1988), 48–57.

[3] BORKAR, S., COHN, R., COX, G., GLEASON, S., GROSS, T., KUNG, H. T., LAM, M., MOORE, B., PETERSON, C., PIEPER, J., RANKIN, L., TSENG, P. S., SUTTON, J., URBANSKI, J., AND WEBB, J. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing '88* (Nov. 1988), pp. 330–339.

[4] BORKAR, S., COHN, R., COX, G., GROSS, T., KUNG, H. T., LAM, M., LEVINE, M., MOORE, 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] CARRIERO, N., AND GELERNTER, D. Applications experience with Linda. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems* (New Haven, CT, July 1988), pp. 173–187.

[6] CARRIERO, N., AND GELERNTER, D. Data parallelism and Linda. In *Proc. 5th Intl. Workshop, Languages and Compilers for Parallel Computing*, vol. 757 of *Lecture Notes in Computer Science*. Springer, 1992, ch. 10, pp. 145–159.

[7] 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.

[8] CHAPMAN, B., MEHROTRA, P., VAN ROSENDALE, J., AND ZIMA, H. A software architecture for multidisciplinary applications: Integrating task and data parallelism. Tech. Rep. 94-18, ICASE, NASA Langley Research Center, Hampton, VA, Mar. 1994.

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

[10] 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), pp. 21–24.

[11] CHOUDHARY, A., NAHARI, B., NICOL, D., AND SIMHA, R. Optimal processor assignment for a class of pipelined computations. *IEEE Transactions on Parallel and Distributed Systems 5*, 4 (1994), 439–444.

[12] CROVELLA, M., AND LEBLANC, T. The search for lost cycles: A new approach to parallel program performance evaluation. Tech. Rep. 479, Computer Science Department, University of Rochester, Dec. 1993.

[13] 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.

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

[15] HIGH PERFORMANCE FORTRAN FORUM. High Performance Fortran language specification, version 1.0. Tech. Rep. CRPC-TR92225, Center for Research on Parallel Computation, Rice University, May 1993.

[16] MCRAE, G., GOODIN, W., AND SEINFELD, J. Development of a second-generation mathematical model for urban air pollution - 1. Model formulation. *Atmospheric Environment 16*, 4 (1982), 679–696.

[17] MCRAE, G., RUSSELL, A., AND HARLEY, R. *CIT Photochemical Airshed Model - Systems Manual*. Carnegie Mellon University, Pittsburgh, PA, and California Institute of Technology, Pasadena, CA, Feb. 1992.

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

[19] RINARD, M., SCALES, D., AND LAM, M. Jade: A high-level machine-independent language for parallel programming. *IEEE Computer 26*, 6 (June 1993), 28–38.

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

[21] 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.

[22] STICHNOTH, J., O'HALLARON, D., AND GROSS, T. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing 21*, 1 (1994), 150–159.

[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., O'HALLARON, D., GROSS, T., DINDA, P., AND WEBB, J. Communication and memory requirements as the basis for mapping task and data parallel programs. In *Proc. Supercomputing '94* (Nov. 1994). To appear.

[25] SUBHLOK, J., STICHNOTH, J., O'HALLARON, D., AND GROSS, T. Exploiting task and data parallelism on a multicomputer. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)* (San Diego, CA, May 1993), pp. 13–22.

[26] 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.

[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.

[29] YANG, B., WEBB, J., STICHNOTH, J., O'HALLARON, D., AND GROSS, T. Do&Merge: Integrating parallel loops and reductions. In *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing* (Portland, OR, Aug. 1993), vol. 768 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 169–183.

[30] YANG, T., AND GERASOULIS, A. Pyrros: Static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 1992 International Conference on Supercomputing* (Washington, D.C., July 1992), pp. 122–129.