Task Parallel Programming in Fx

Jaspal Subhlok, David R O'Hallaron and Thomas Gross

CMU-CS-94-112

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Many important applications have a heterogeneous structure, and can be implemented efficiently only with task parallelism. This paper presents a set of extensions to Fortran to build task parallel programs for multicomputers consisting of distinct nodes: either private memory parallel machines, or autonomous computers connected by a high speed network. The design of these extensions is driven by the following objectives:

- The compiler should handle inter-node communication, not the programmer.
- The tasking extensions should be integrated into an existing parallel programming language, so that existing programs can benefit from task parallelism, existing libraries can be used, and the users do not have to learn a new language.
- Current compiler technology should be able to produce efficient parallel programs that are competitive with hand parallelized codes.

Our implementation of task parallelism is integrated with an HPF like data parallel Fortran compiler (Fx) developed at Carnegie Mellon University. The design, implementation, and experimental results from Fx are presented in several related publications[SSOG93, SOGP94, SOG93, YWS+93]. In this paper, we describe the programming model and the set of Fortran extensions for task parallelism and discuss the basis for the design decisions.

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, program mapping, multparadigm programming

1 Introduction

In the past few years, compilation of programs for private memory parallel computers has received considerable attention. High Performance Fortran(HPF) has emerged as a standard extension to Fortran for parallel computing. This development allows programmers to write and compile data parallel programs for a variety of architectures, but HPF is not suitable for *task parallel* or *heterogeneous* computing.

Heterogeneity can be introduced due to the following two reasons:

- Applications: An application is often composed of a set of different modules, with data parallelism inside modules, and task parallelism between them.
- Computing environments: Use of different computers over a high speed network is an increasingly popular way of parallel computing. Individual computers can themselves be parallel computers, and can vary from workstations to supercomputers.

It is clear that it is important to support task parallelism alongside a compilation and execution environment for data parallel programs. However, there are many ways in which this can be accomplished. Since dealing with the details of the inter-node communication is cumbersome and error prone, we have taken the approach that all communication should be generated by the compiler, and the user writes programs for a common data space. This approach also allows communication optimizations by the compiler. Our design of task parallelism is in the context of a data parallel compiler, and our implementation is integrated with a prototype High Performance Fortran compiler; task parallelism is expressed using additional *directives*. 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 easy to convert existing programs to task and data parallel programs, and it is easier to find user acceptance. Finally, it is important to be able to compile task and data parallel programs efficiently using existing compiler technology. If the compiler generated programs are not competitive with programs written by hand, it is very difficult to gain user acceptance. In particular, we allow several directives to help the compiler in generating efficient code, even though some of them may become obsolete as sophisticated compilers become available.

Our current implementation is for an iWarp parallel machine, and a PVM based implementation for heterogeneous networks is being developed. 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 [SOGP94].

This paper is organized as follows. In Section 2, we present the programming model. In Section 3, we present the compiler directives for task parallelism, and illustrate them with an example. We discuss some of the compilation issues in Section 4. In Section 5 we discuss our rationale for the design decision that we have made. Section 6 discusses future developments integration with HPF.

2 Programming model

We describe the basic programming model. A parallel program consists of a set of (possibly data parallel) *task-subroutines*. A *task* corresponds to the execution of one call to a task-subroutine. Thus, the granularity of task parallelism is a single procedure invocation. Task-subroutines are data parallel subroutines with well defined side effects. To ease the task of the compiler, the programmer includes directives that specify the side effects of the subroutines, with respect to the variables they reference as actual parameters.

A program begins execution as a single (data parallel or sequential) task on one machine. New tasks are created by calls to task-subroutines. Since the data relationship of the calling program to the task-subroutine calls is well defined, the compiler can map the tasks on different sets of processors, and generate communication to maintain data consistency. The basic paradigm is that the results obtained should be consistent with those obtained with sequential execution.

The mapping of tasks onto processing nodes is a critical factor in obtaining good performance, although it does not effect correctness. In the simplest case, each task-subroutine is mapped to a distinct set of nodes which execute all tasks created by the calls to that task-subroutine. A task-subroutine can be *replicated* and mapped to multiple sets of nodes, in which case successive calls to this task-subroutine are executed by different instances of the task-subroutine on different sets of nodes, in round robin fashion. Multiple task-subroutines can be mapped to the same set of nodes (this collection is then called a *module*), but only one task-subroutine is active at any given time on a single node.

The programmer can control the mapping using directives. We are also developing tools to automatically choose an efficient mapping, and to assist the programmer in choosing one [Sub93]. In this paper, we assume that the mapping process is driven by explicit directives, which may be provided by the programmer, or automatically generated by a tool. The compiler currently supports mappings that are fixed at compile time, since that is simpler and requires minimal support from the data parallel compiler and the runtime system. Support for a dynamic implementation is also discussed.

The main characteristics of the programming model can then be summarized as follows:

- No new language constructs, only special compiler directives.
- Common name space for shared data.
- Data parallel subroutines with well defined side effects are units for task parallelism.
- Communication between tasks is generated and managed by the compiler.
- Sequential consistency, determinism, and freedom from deadlock guaranteed by the compiler.

3 Compiler Directives

We have not introduced any new language features, and rely entirely on compiler directives for expressing task parallelism. Some directives are required by the compiler for generating a correct task parallel program, while others are used to guide the compiler in making performance related decisions like program mapping. We state and explain the directives that are used to express task parallelism, and illustrate them with an example. We will discuss the limitations and possible extensions to the directives in a later section.

3.1 Parallel section

Calls to task-subroutines are permitted only in special code regions called *parallel sections*. 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 processors) efficiently at compile time. There can be other directives in a parallel section that specify the input/output behavior and resource requirements of task-subroutines, which are discussed later. A parallel section corresponds to a mapping of task-subroutines to processors and other resources. On exit from a parallel section, the program reverts to the mapping prior to the beginning of the parallel section. Subsequent parallel sections may have different mappings.

These restrictions are necessary to make it possible to manage shared data and shared resources (including processors) efficiently at compile time.

A `begin parallel`...`end parallel` pair specifies a parallel section. Statements inside a parallel section must be task-subroutine calls, loop headers, or loop delimiters.

```
C$   begin parallel
        .
        .
     body consisting of task-subroutine
     calls, loops and other directives
        .
        .
C$   end parallel
```

3.2 Input and output parameters

Calls to task-subroutines have `input` and `output` directives that precisely define the side-effects of the subroutine call, that is, the data space that the subroutine accesses and modifies. Every variable in the calling program, whose value at the call site may potentially be used by the called subroutine, must be added to the input parameter list. Similarly, every variable in the calling program, 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.

```
C$   begin parallel
        .
        .
     call foo(a,b,i)
C$   input a(:,:), i
C$   output b(:,1)
        .
        .
C$   end parallel
```

3.3 Directives for mapping programs

The programmer controls the placement of task-subroutines onto sets of processors and machines with directives. For a homogeneous processor array, 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. Different task-subroutines can be mapped together or separately. Multiple locations can be given for a single task-subroutine, which will generate replicated instances of the task-subroutine, that execute the calls to the task-subroutine at different locations, in round robin fashion.

We currently support two directives. The `processors` directive states how many processors should be assigned to the task-subroutine. The `origin` directive states the location(s) for the task-subroutine in the processor array. In the implementation, only rectangular subarrays can be assigned to task-subroutines, and the processor array is in two dimensional space. Hence, both directives have a pair of integers as argument.

```
C$   begin parallel
        .
        .
     call foo(a,b,i)
C$   processor (4,4)
C$   origin (0,0)
        .
        .
C$   end parallel
```

3.4 An example program

We will use the program in Figure 1 to show how the directives discussed above are used to write a task parallel program and guide it's mapping. The program contains a *src* routine that supplies data to routines *p1* and *p2* for processing, which in turn send their output to a *sink* routine.

The task-subroutine *src* has variables *A* and *B* as output parameters, task-subroutines *p1* and *p2* have *A* and *B* as there input-output parameters, respectively, and task-subroutine *sink* has both *A* and *B* as input parameters. Using sequential execution as the basis to match inputs and outputs of subroutines, the compiler constructs the task dependence graph shown in Figure 1. Task-subroutines *src* and *p1* are mapped to the same module (M1) since they have identical *origin* directives. Similarly, task-subroutines *p2* and *sink* are mapped together to another module (M2). Based on the *processor* and *origin* directives, the modules are mapped as shown in Figure 1. Since the task-subroutines in M2 have two arguments to the *origins* directive, the module is *replicated*. Successive invocations of the corresponding task-subroutines are executed alternately on the two different instances of the module.

4 Compilation of task parallelism

The compiler has to perform a set of steps to support task parallelism, which can be summarized as follows:

1. Identify the task structure of the program and determine the placement of task-subroutines (mapping).

```
C$    begin parallel
      do i = 1,10
         call src(A,B)
C$       output (A,B)
C$       processor (2,4)
C$       origin (0,0)
         call p1(A)
C$       input (A), output (A)
C$       processor (2,4)
C$       origin (0,0)
         call p2(B)
C$       input (B), output (B)
C$       processor (2,2)
C$       origin (2,0), (2,2)
         call sink(A,B)
C$       input:  (A,B)
C$       processor (2,2)
C$       origin (2,0), (2,2)
      enddo
C$    end parallel
```

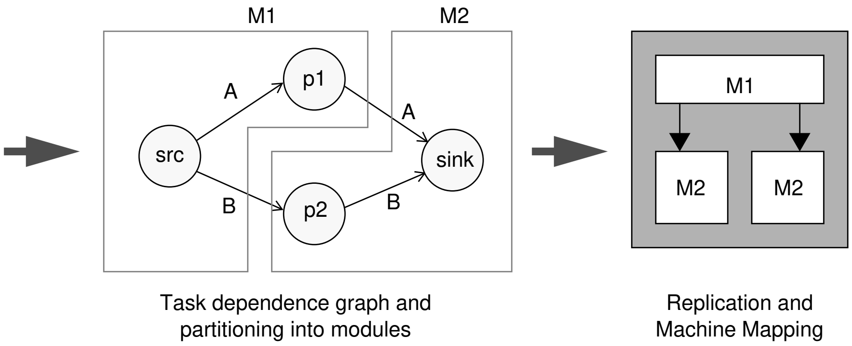

Figure 1: Compilation of task parallelism

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.

One of the primary goals of our selection of a language for expressing task parallelism is that it should be easy to compile to efficient parallel programs. In the rest of this section, we discuss some parts of the compilation process to illustrate how the choice of directives has led to a fairly simple compilation scheme.

4.1 Mapping programs

The mapping of a program consists of a sequence of steps. First, the task-subroutines are partitioned into modules. All task-subroutines in the same module are mapped to the same set of processors. Second, the modules may be *replicated* to form multiple instances of modules. Different module instances execute on different sets of processors, and the compiler uses a round-robin schedule to determine which module instance executes a particular task-subroutine call. Finally, a set of machine nodes is assigned to each module instance.

The set of legal mappings is determined by resource constraints, and selecting a good mapping is important for obtaining good performance. However, the selection of a mapping is not a part of the core compiler, which assumes that the mapping information is provided by the user or another tool. The program characteristics that influence the choice of a mapping are discussed in [SOGP94], and an automatic mapping scheme is discussed in [Sub93].

4.2 Array section analysis

The compiler has to precisely determine the the data elements that must be communicated between task-subroutines. The input and output directives determine the variables that a task-subroutine call defines and uses. Since conditional statements are not permitted inside a parallel section, there is a unique definition that reaches a use, and this can be determined using data flow analysis. The main complexity comes from the fact that definitions and uses can be of array sections, not just scalar variables.

Because of the constraints on the input and output directives, only constant rectangular array sections can be specified as input or output parameters. The compiler has to be able to perform union and subtraction of array sections for data flow analysis, but these are straightforward for rectangular sections. In future, we plan to relax some of the constraints, and use more sophisticated analysis. However, some constraints are necessary for compile time analysis of data movement between task-subroutines.

4.3 Communication generation and scheduling

Once the compiler has determined the task-structure and mapping of the program, and the data that needs to be transferred between task-subroutines, the actual communication code is generated. The compiler also determines the data distribution of array parameters inside task-subroutines, so that parameters that are sent from other modules are distributed correctly. The actual communication code is external to the task-subroutines and ensures that the subroutines have data in the expected distribution on entry.

The communication primitives used obviously depend on the primitives that are available, but the compiler may have to make some important choices. In our implementation, two different schemes are possible; the first based on systolic communication and the second based on message passing communication. In the first scheme, all processors in a module are connected in a ring, and the data is routed systolically over a single channel. This scheme provides extremely low latency, low overhead communication with no buffering. Another advantage of this scheme is that the sending module needs no knowledge of the receiving module, and simply puts the data on a channel in a canonical order. In the second scheme, the sending module computes the processor number and address for each data element that has to be sent, and sends it using a message passing library. This method can use parallelism in communication, although it has a somewhat higher overhead than the previous scheme. The compiler selects one of the communication schemes for each communication step based on the granularity of communication, or the user can explicitly select one scheme using a compiler flag.

The compiler also has to find a global communication schedule that guarantees freedom from deadlock. When a task-subroutine has to send data to multiple tasks corresponding to multiple task-subroutine calls, the data is sent in the order in which those subroutine calls would have executed in the original sequential program. A similar ordering scheme is used for receiving data. It is easy to see that this guarantees deadlock freedom, since when no task is executing, the unexecuted task corresponding to the earliest execution in the sequential program, must be ready to execute.

5 Discussion

The approach that we have taken towards task parallelism in a data parallel environment is characterized by the following features:

- Task parallelism is expressed by specifying task parallel subroutines and their side effects. There are no explicit communication statements in the user's program.
- The compiler does most of the management of task parallelism, and the runtime overhead is minimal. To achieve this, it is necessary to place certain restrictions on the computations that can be expressed in our model.
- No new programming constructs are added to the language. The directives are critical to the performance, but do not effect the semantics.

Our basic goal is to provide support for task parallelism in a way that would require minimal change in the user program (which may be sequential or data parallel), and can be implemented efficiently with relatively simple compiler technology. In this respect, our approach is similar to that taken in HPF. We now discuss and justify the main design decisions that we have made, in the context of alternatives that were available.

5.1 User or compiler generated communication

We have taken the view that all communication for task parallelism should be generated by the compiler. This is analogous to the approach to data parallelism taken in High Performance Fortran. An alternate approach is taken in Fortran M [FC92], where the programmer uses a portable message passing interface for communication between task parallel components.

Compiler generation of communication has several advantages. The most obvious and important is that the user does not have to actually write the communication code between tasks, which can be a difficult and error-prone task.

Only a high level definition and use information is needed. This makes it relatively easy to convert existing sequential or data parallel programs to programs with task parallelism.

Portability is another important and desirable property. When using explicit communication, portability can be achieved by using a portable communication layer. This approach can also be used when the compiler is generating the communication, i.e the compiler can generate calls to a portable communication library. In addition, the compiler can also use any other communication mechanisms that may be available. The compiler can also perform optimizations relating to choosing communication primitives and scheduling communication because of global knowledge of data movement.

To make it possible for a compiler to generate efficient communication with minimal analysis, we have imposed some restrictions on how task parallelism is expressed. For instance, data transfer is permitted only on entry and exit from a task-subroutine. Also, input and output parameters must be variables, whole arrays, or simple array sections, but arbitrary array sections are not allowed. While some of these restrictions can be relaxed if more sophisticated analysis was used, it is clear that some restrictions are necessary. This implies that the user does not have fine control over parallel execution, as he or she would with explicit message passing. We believe that this price has to be paid for using a high level model for task parallel computations.

5.2 Static or dynamic scheduling

In our compiler, placement and scheduling of tasks is done at compile time. It is possible to postpone some or all of these decisions to runtime. For example, in Jade [LR91] task scheduling is done completely at runtime.

Management of tasks at compile time has some advantages. The overhead of parallel execution is minimized and only a simple runtime system is needed. The individual data parallel subroutines can be compiled more easily and efficiently if the number of processors they are to execute on is fixed at compile time.

It is clear that there are many applications that would benefit from dynamic scheduling of tasks. In particular, dynamic load balancing is important for many task parallel applications. The basic programming model can support a dynamic implementation, and the reason for a static approach is efficiency and simplicity. In future we plan to extend the directives and the compiler to be able to support more dynamic programs, and also learn from the experience of other researchers.

However, our research is directed towards programs whose main components are data parallel, but need task parallelism for good performance. We do not expect to efficiently support programs that create and destroy tasks dynamically as the basic mechanism for creating parallelism.

5.3 Language constructs or compiler directives

We do not introduce any new language constructs, and instead use compiler directives to specify and manage parallelism. We believe that this property is very important for building task and data parallel programs from existing data parallel or sequential programs. In particular, except for compiler directives, identical programs can be used for sequential, purely data parallel, and various task parallel implementations of a program.

While there is less freedom in choosing how to express task parallelism without defining new language constructs, we do not think it is a handicap for our purpose, which is to make it possible to exploit task parallelism in a data parallel environment. The situation is similar to HPF where (almost) all the parallelism related information is expressed as directives without changing the language.

6 Extension and standardization of task parallelism

Use of subroutines (or any program components) with well defined side effects is a powerful paradigm for heterogeneous task parallelism. We believe that the set of directives described in this paper, or some variation of them, would be a valuable addition to a language like HPF. There are several ways in which the directives can be extended to make the language more powerful. However, when considering extensions, it is important to keep in mind that the directives must lead to a simple, portable and efficient compilation scheme.

Making the directives more general can make it more difficult for the compiler to generate efficient parallel programs and/or make it necessary to postpone some analysis to runtime. We discuss some of the extensions that we are considering, and comment on their importance and impact on implementation.

Reducing restrictions on input and output directives

We currently permit array sections in input and output directives, but each dimension and the step size must be a known constant, or "colon" implying all elements in that dimension. It will certainly be useful to relax some of these restrictions.

Allowing ranges in dimensions (rather than a single constant) would certainly be useful, and also relatively easy to implement. Allowing variables in array specifications would also be useful, but will have a significant impact on the compiler. The compiler will not be able to statically determine which task-subroutines communicate, and what data is to be communicated. In future, we plan to allow simple loop index expressions in array section specifications, but not arbitrary expressions. The objective is to allow more general array section specifications, while ensuring that the compiler is able to generate efficient communication code.

Nested task parallelism

We do not permit nested task parallelism. In our experience, nested task parallelism is less important when data parallelism is also available. However, more experience with task parallel applications is needed before the importance of nested parallelism is well understood.

General directives for resource management

We support directives to allocate processors to task-subroutines and decide their placement. In general, the compiler will need more information to map programs, particularly in a heterogeneous environment. For example, the compiler will need to know what kind of computing and I/O resources have to be made available to different task-subroutines. The specific information that is needed will be dependent on the environment.

Our approach is to allow a general directive that would list all the resources that are needed by a task-subroutine, which may be machine dependent. The host environment will provide the compiler with a default list of available resources, which may be modified by a user. The entries that are permitted in this directive, and the effect they would have, may be dependent on the execution environment.

Integrating with Fortran 90

We have developed the task parallelism directives in a Fortran 77 environment. Fortran 90 has some added features to support modular programming, which may be used to build an improved interface to task parallelism. For example, modules can be used to declare blocks of data that are shared by a subset of task-subroutines but are not global, thus making data management easier for the compiler. However, we have not examined the potential of using these features for task parallelism in detail.

7 Conclusions

We have presented a set of compiler directives which can be combined with a data parallel language like High Performance Fortran to write task and data parallel programs for a private memory parallel machine, or a heterogeneous computing environment. We believe that the approach is a simple and portable way of developing efficient task and data parallel programs.

References

[FC92] I. Foster and K. Chandy. Fortran M: A language for modular parallel programming. Technical Report MCS-P327-0992, Argonne National Laboratory, June 1992.

[LR91] M. Lam and M. Rinard. 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.

[SOG93] J. Stichnot, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, Aug 1993.

[SOG+94] J. Subhlok, D. O'Hallaron, T. Gross, P.Dinda, and J. Webb. Communication and memory requirements as the basis for mapping task and data parallel programs. Technical Report CMU-CS-94-106, School of Computer Science, Carnegie Mellon University, January 1994.

[SSOG93] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[Sub93] J. Subhlok. Automatic mapping of task and data parallel programs for efficient execution on multicomputers. Technical Report CMU-CS-93-212, School of Computer Science, Carnegie Mellon University, November 1993.

[YWS+93] B. Yang, J. Webb, J. Stichnoth, D. O'Hallaron, and T. Gross. Do&merge: Integrating parallel loops and reductions. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, Aug 1993.