Task Parallelism and High-Performance Languages

Ian Foster
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

Abstract

High-Performance Fortran as currently defined cannot be used to solve all programming problems. However, its focus on regular problems and data-parallel algorithms stems not from a belief that it is only these problems that matter, but rather from the fact that it was in this area that it was easiest to build consensus as to what was required in a language for high-performance computing. In future work, this initial consensus will have to be extended. One direction to be considered in the next round of HPF Forum meetings is task parallelism. In this paper, we examine and illustrate the considerations that motivate the use of task parallelism. We also describe one particular approach to task parallelism in Fortran, namely the Fortran M extensions. Finally, we contrast Fortran M with other proposed approaches and discuss the implications of this work for task parallelism and high-performance languages.

1 Introduction

The definition of High Performance Fortran (HPF) is a significant event in the maturation of parallel computing: it represents the first parallel language that has gained widespread support from vendors and users. If successful, it will advance parallel computing by simplifying application development, increasing application portability, and providing a common framework for both commercial and academic research and development of compilers, debuggers, performance analyzers, and the other tools that are currently lacking on parallel computers. HPF has also stimulated similar developments in other languages, notably pC++ [6]. These languages are commonly referred to as *high-performance languages*, because they are designed to allow efficient compilation for high-performance parallel computers.

HPF as currently defined cannot be used to solve all programming problems. Indeed, its focus on regular problems and data-parallel algorithms makes it dangerously limited. However, this focus stems not from a belief that it is only these problems that matter, but rather from the fact that it was in this area that it was easiest to build consensus as to what was required in a Fortran-based language for high-performance computing. In future work, this initial consensus will have to be extended to encompass other areas. This process has already started with the second round of HPF Forum meetings.

One promising direction is to define additional directives that can be used to make more information available to the compiler. Armed with this information, the compiler can parallelize programs that operate on irregular and adaptive data structures, or that require pipeline structures for efficient parallel execution. The goal here is to generalize rather than to abandon the single-program, multiple-data (SPMD) programming model of HPF.

Another important direction, which is the subject of this paper, is to incorporate support for task parallelism. The term task parallelism (sometimes called control or functional parallelism) refers to the explicit creation of multiple threads of control, or tasks, which synchronize and communicate under programmer control. Task and data parallelism are complementary rather than competing programming models. While task parallelism is more general and can be used to implement algorithms that are not amenable to data-parallel solutions, many problems can benefit from a mixed approach, with for example a task-parallel coordination layer integrating multiple data-parallel computations. Other problems admit to both data- and task-parallel solutions, with the better solution depending on machine characteristics, compiler performance, or personal taste. For these reasons, we believe that a general-purpose high-performance language should integrate both task- and data-parallel constructs. The challenge is to do so in a way that provides the expressivity needed for applications, while preserving the flexibility and portability of a high-level language.

In this paper, we examine and illustrate the considerations that motivate the use of task parallelism. We also describe one particular approach to task parallelism in Fortran, namely the Fortran M extensions. Finally, we contrast Fortran M with other proposed approaches and discuss the implications of this work for task parallelism and high-performance languages.

2 High Performance Fortran

We first provide a brief review of HPF [8]. As currently defined, this is primarily a data-parallel language, meaning that it allows programmers to exploit the concurrency that derives from the application of the same operations to all or most elements of large data structures. An HPF program is a sequence of such operations, which may be specified using either explicitly parallel constructs (e.g., array expressions and `FORALL`) or implicitly using traditional `DO` loops. A program can be augmented with distribution directives specifying how data is to be mapped to processors. An HPF compiler normally generates a single-program, multiple-data (SPMD) parallel program by applying the "owner computes" rule to partition the operations performed by the program; the processor that "owns" a value is responsible for updating its value. The compiler also incorporates communication operations when computation assigned to one processor requires data located on other processors.

In addition to its data-parallel constructs, HPF provides two mechanisms for specifying task-parallel execution. The first is the `PURE` function, which when called within a `FORALL` loop can perform different computations depending on the value of its arguments. However, the utility of this construct is limited by the fact that concurrently-executing function calls cannot communicate. The second mechanism is the `EXTRINSIC` function, which creates a separate execution thread on each processor. These concurrent threads have access to the data structures of the HPF program and can perform arbitrary computation and communication. This is a generic escape mechanism that allows an HPF program to call a message-passing library to perform arbitrary task-parallel computation; however, it lacks flexibility and modularity and is not an adequate mechanism for general-purpose task-parallel programming.

An important advantage of the current HPF model is that programs have sequential (single-threaded) semantics. A program that adheres to the HPF standard and that does not use extrinsic functions can be read as if it were a sequential Fortran program; the parallel constructs and distribution directives affect performance but not correctness. In principle, this feature simplifies program development and debugging, as many concepts and tools from sequential programming can be reused. It is unclear to what extent this sequential semantics can be maintained as HPF is extended to address a wider range of problems.

3 The Role of Task Parallelism

The data-parallel constructs incorporated in high-performance languages exploit a common attribute of computations in science and engineering, namely *homogeneity.* Computations that apply the same operation to all elements of regular data structures can be expressed elegantly using data-parallel constructs and can be compiled efficiently for parallel computers. For these problems, there is plentiful evidence that data parallelism is an effective solution [7].

Unfortunately, *heterogeneity* — whether in data structures, computation, or data dependencies — appears to be equally prevalent, particularly as high-performance computers are used to solve more complex problems and as scientists and engineers use more sophisticated algorithms. Heterogeneity does not prevent the use of data parallelism; however, depending on the degree of heterogeneity, a data-parallel program may be overly complicated (because the programmer is forced to use inappropriate abstractions) or inefficient (because the compiler, lacking information available to the programmer, cannot infer efficient execution and communication schedules). The programmer may then find it simpler or more efficient to specify parallel algorithms explicitly, using task-parallel constructs.

In this section, we examine what appear to be the principal considerations motivating the use of task parallelism, illustrating each with examples. These are as follows:

1. Software Engineering. Independent of issues relating to parallel computing, there may be software engineering benefits from treating separate programs as independent tasks. These benefits may relate to modularity or to the need to execute in a heterogeneous system.

2. Locality. Task parallelism can allow the programmer to enhance locality and hence performance by executing different components of a problem concurrently on disjoint sets of processors.

3. Scheduling. Task parallelism can allow the programmer to enhance performance by specifying computation and communication schedules that could not be discovered by a compiler.

3.1 Software Engineering

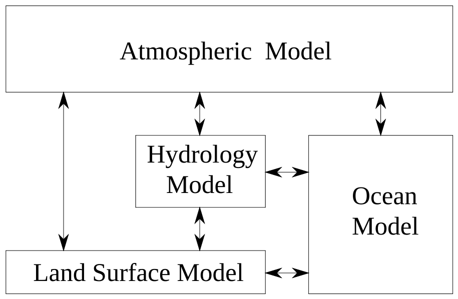

Figure 1: An environmental modeling system, incorporating four submodels; a realistic model might incorporate many more components. Arrows represent data transfer.

In the first set of applications that we consider, the use of task parallelism is motivated primarily by software engineering concerns, rather than performance: in particular, the need to construct complex parallel programs in a modular fashion or to execute different program components on different computers in a heterogeneous network.

These concerns often arise in *multidisciplinary simulations*, in which a computer model of a complex system (such as an automobile, regional air quality, or an electricity distribution system) is constructed from models of system components (Fig. 1). The system model may itself form part of a design or management system that incorporates optimizers, databases, external control functions, etc. In principle, the various component models could be integrated into a single (data-parallel) program, which would call components as subroutines. However, this approach suffers from a lack of modularity. Program development, validation, and modification all tend to be more difficult when program components are tightly interconnected, particularly if the components were not originally designed to be integrated in this manner.

An alternative approach is to treat the components of a multidisciplinary simulation as separate programs that execute concurrently and exchange information using an "arms-length" mechanism such as message passing or files: in other words, to structure the application as a task-parallel program. Interfaces between components are then simple and well defined, and components can be executed on different computers if required. Components can also be executed in parallel to improve performance; however, this is not a prerequisite for task parallelism to be useful.

In simple problems of this sort, there are a fixed number of component models that can themselves be expressed as data-parallel programs. Hence, heterogeneity is restricted to an outer "coordination layer" which orchestrates the execution of data-parallel computations. In more complex problems, subcomputations may themselves be heterogeneous in structure.

3.2 Locality

A second motivation for the use of task parallelism is to enhance locality so as to make more efficient use of resources such as cache, memory, or communication bandwidth. This is achieved by executing different parts of a problem on disjoint subsets of available pro-

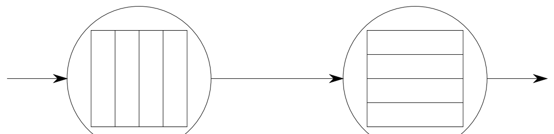

Figure 2: Task-parallel computation in a two-dimensional FFT. Two subcomputations execute as a pipeline, with processors in the first phase operating on columns of each array, and processors in the second phase operating on rows.

cessors. (A side effect is to increase available concurrency; this can be equally important in some problems.)

This situation often arises in signal processing when applying sequences of transformations to streams of pixel arrays. Data sets cannot easily be scaled as the number of processors increase, and different inputs cannot be transformed independently. Hence, while a purely data-parallel solution is possible, in which the various transformations are applied sequentially, communication costs can often be reduced significantly by computing each transformation on a separate set of processors. Different inputs can then flow through this pipeline concurrently. Each transformation can be performed by a task- or data-parallel program.

A simple example of a pipelined computation is a two-dimensional fast Fourier transform (2-D FFT). When applied to an array of size $N \times N$, this performs first N independent 1-D FFTs, one on each column of the array, and then a second set of N independent 1-D FFTs, one on each row. A data-parallel solution might initially decompose the array by columns so that the first set of 1-D FFTs could proceed without communication, then transpose the array before performing the second set of FFTs. A task-parallel solution might construct a pipeline of two stages, with the first stage performing FFTs on columns and the second FFTs on rows (Fig. 2). The same amount of data must be communicated in both cases: in the data-parallel program, the entire array is transposed, while in the task-parallel program, the entire array is forwarded in the pipeline. However, the task-parallel program sends only about one-quarter as many messages, because rather than each of the P processors sending to each other processor (in the transpose), each of the $P/2$ processors in the first pipeline stage must send only to the $P/2$ processors in the second stage. This can improve overall performance.

Other problems in which locality can motivate the use of task parallelism include multiblock and nested grid codes, and (in many cases) multidisciplinary simulations.

3.3 Scheduling

In a third class of applications, the use of task parallelism is motivated by a need to improve performance by explicitly controlling the scheduling of computation and communication operations. This requirement arises frequently in applications with irregular, data-dependent computation and communication requirements.

Data-parallel language compilers have proven to be most successful when computa-

```
do i = 1,ni
  do j = 1,nj
    do k = 1,nk
      do l = 1,nl
        I = compute_integral(i,j,k,l)
        F(i,j) = F(i,j) + I*D(k,l)
        F(k,l) = F(k,l) + I*D(i,j)
        F(i,k) = F(i,k) + I*D(j,l)
        F(i,l) = F(i,l) + I*D(k,l)
        F(j,l) = F(j,l) + I*D(i,k)
        F(j,k) = F(j,k) + I*D(i,l)
      enddo
    enddo
  enddo
enddo
```

Figure 3: Logic for Fock matrix construction problem

tion and communication requirements are either predictable at compile time or easily determined at run time. This predictability allows the compiler and/or runtime system to determine a static execution schedule for each processor. In more heterogeneous problems, the computation and communication performed by a program are irregular, either in time or space, and a compiler may not be able to discover an efficient mapping of computation to processors, scheduling of computation within processors, and organization of communication between processors. However, a programmer may be able to use task-parallel constructs to implement effective application-specific mapping and scheduling strategies. A task-parallel problem formulation can also underspecify scheduling constraints, allowing the use of data-driven execution models in which the ordering of computation on each processor is determined by the availability of data.

A simple problem of this sort is the Fock matrix construction problem from computational chemistry. The core of this problem is the quadruple-nested loop illustrated in Fig. 3. Approximately N^4 integrals must be computed; each requires data from six elements of a density matrix, D, and contributes to six elements of a Fock matrix, F. Both D and F have size $N \times N$ and must be distributed. The cost of an integral is strongly data dependent. An efficient parallel algorithm must both map integrals to processors dynamically and block integrals and communications so that fewer than $6N^4$ messages are required to communicate the D and F values.

A data-parallel formulation of this problem is possible if the data-parallel language provides an "accumulate" operation. However, the problems of mapping and blocking integrals remain. A task-parallel solution can define distinct compute and data server threads, placing one thread of each type on each processor (Fig. 4). The compute threads perform computation and generate requests for data to the data server threads, while the data server threads handle requests for data. The desired behavior is that a compute thread executes when no requests are pending, and that execution switches to a data

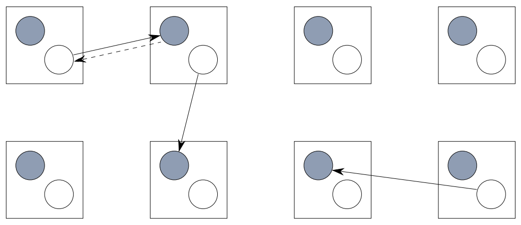

Figure 4: Task-parallel computation in a Fock matrix computation. Each box represents a processor; distinct compute tasks (represented by unshaded circles) and data server tasks (shaded) interact to provide asynchronous access to distributed data structures.

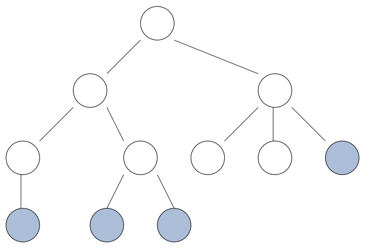

Figure 5: Task parallelism in a search problem. Each leaf node evaluation (shaded) involves a data-parallel computation; the shape of the tree is determined only at run time.

server thread when a request arrives. The scheduling of computation and communication is then under programmer control. In particular, integrals can be allocated to compute threads using a load balancing strategy, and compute threads can block requests to data server threads.

As a second example, we consider search problems in which a search tree of unknown size and shape must be explored, with an evaluation function applied at each node and pruning used to limit the number of nodes explored (Fig. 5). In this case, both the communication and computation structure of the application are irregular. Efficient parallel algorithms typically use dynamic load balancing algorithms to allocate computation to processors. In some situations, the evaluation function can be sufficiently complex to benefit from a data-parallel formulation, in which case the problem involves a task-parallel collection of data-parallel computations.

Other examples of problems in which programmer control of scheduling may be necessary for performance are discrete event simulation, reactive computations that must respond to input from users or sensors, and adaptive mesh refinment problems, in which meshes may be created, destroyed, or moved over time. We can also point to multidisciplinary and image-processing applications in which the amount of computation in each component, or the structure of the computation, adapts during program execution.

3.4 Summary

This brief examination of task-parallel applications has suggested at least three reasons for using task parallelism in addition to data parallelism: software engineering, locality, and scheduling. Of course, we can also identify problems in which several of these motivations apply. For example, there can be software engineering advantages to formulating a multigrid code as a task-parallel collection of interacting data-parallel programs; task parallelism can also be used to improve performance by executing multiple grids concurrently to improve locality, and by scheduling these computations efficiently.

4 The Fortran M Approach

The Fortran M (FM) extensions to Fortran [5] represent a particularly simple approach to task parallelism in Fortran. FM has also been used as a task-parallel coordination language for HPF [4].

FM is a small set of extensions to Fortran 77 (F77) for specifying concurrent execution, communication, synchronization, and resource management. A major design goal was to define extensions consistent with F77 concepts. This means, for example, that because F77 lacks structured data and dynamic memory allocation, these concepts are not used in the extended language. As we note below, the richer set of constructs available in Fortran 90 (F90) can enable more flexible approaches. The FM extensions can be characterized as follows:

1. Processes, modeled on F77's subroutine construct, provide the basic building block from which parallel programs are constructed. They encapsulate data and the code that operates on that data.

2. Parallel block and parallel do-loop constructs are used to create instances of processes.

3. Processes can communicate and synchronize both by passing data to subprocesses and by sending and receiving data on channels. The channel operations are modeled on F77's file I/O constructs, and indeed channels can be thought of as "virtual files."

4. Both processes and channels can be created and deleted, and channels can be reconnected, dynamically; nevertheless, a compiler and runtime system can enforce deterministic execution.

5. Resource management constructs allow the programmer to control how processes are mapped to processors. These constructs are modeled on F77's array constructs: a programmer can define a virtual processor array, locate processes within this array, and invoke subcomputations on subarrays.

It should be clear that the extensions are extremely simple: they are modeled mostly on existing Fortran ideas, with the main conceptual extension being dynamic process and channel creation. Nevertheless, they have proved sufficient for a wide variety of problems and, in addition, provide modularity properties needed to support the definition of parallel paradigm libraries, as described below. (The M in FM stands for "Modular".)

```
program aerodynamics
processors p(128)
inport (integer, real x(100,200), real y(100,200)) pi, qi
outport (integer, real x(100,200), real y(100,200)) po, qo
...
channel(in=pi,out=po)
channel(in=qi,out=qo)
...
processes
   processcall controls(pi,qo) submachine(p(1:64))
   processcall structures(qi,po) submachine(p(65:128))
endprocesses
end


process controls(inp,outp)
processors p(64)
inport (integer, real x(100,200), real y(100,200)) inp
outport (integer, integer, real x(100,100)) outp
...
send(outp) i, a, b
receive(inp) nstep, u, v
...
end
```

Figure 6: Sketch of an FM multidisciplinary program

Figure 6 illustrates the use of several FM constructs. The main program uses the `channel` statement to create two channels; a process block (delineated by `processes` and `endprocesses` statements) is used to create processes called `controls` and `structures`. The new processes execute concurrently, with the `submachine` annotations specifying that they should execute on disjoint sets of 64 virtual processors. The second code fragment implements the `controls` process; it uses the `send` and `receive` statements to send and receive data on the ports passed as arguments. Message formats are defined by the port declarations, allowing an FM compiler to generate efficient communication code and to reformat data in a heterogeneous environment. Although `controls` has been invoked on a submachine of 64 processors, it is defined here to execute sequentially; a parallel block (or a call to a message-passing or HPF procedure) would be required to invoke parallel execution.

4.1 Paradigm Integration

In both the multidisciplinary simulation and image processing problems presented as motivating examples, we pointed out that programs can usefully be structured as task-parallel collections of data-parallel programs. These are simple examples of problems that can benefit from the use of multiple parallel paradigms. In other situations, it can be useful to integrate program components developed using message-passing libraries (e.g., linear algebra libraries) or using shared-memory models (e.g., distributed data structures). Integration can be achieved using a single language that combines all the various paradigms. However, this approach tends to be complex. Alternatively, we can define a language that provides just a few basic mechanisms, which we then use to develop libraries implementing the different paradigms. In this section, we outline how FM can be used as a framework of this sort.

The key to using FM to implement multidisciplinary frameworks of this sort is its support for *compositionality*. FM processes specify concurrency, synchronization, communication, and process mapping with respect to logical resources (processes, channels, virtual processors) rather than physical resources. This means that an FM process can be reused in different situations without concern for its internal concurrency, communication, and mapping; hence, we can develop separate program components using libraries implementing message-passing libraries, distributed shared data structures, etc., and then compose these components to form a complete program.

We use a simple example to illustrate how this integration is achieved. A message-passing (MP) compatibility library allows MP programs to be invoked from FM and permits MP programs to call FM routines. An invoking FM program uses FM virtual computer constructs to specify the resources (virtual processors) available to the MP program; the MP program executes as if these virtual processors were physical processors, and performs ordinary MP calls, which, however, are implemented by an FM library rather than direct message passing. (Experimental studies indicate that the overhead of this approach is small.) FM data structures can be passed as arguments to the MP program. These data structures can be either replicated or partitioned over virtual processors. Ports passed as arguments allow an MP program to communicate with other program components. The following code fragment illustrates some of these ideas.

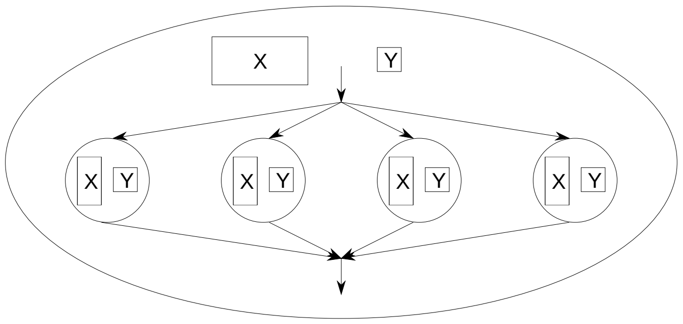

Figure 7: Invoking a message-passing program from FM

```
process execute_mp(X,Y)
processors p(4)
real X(512,4), Y
processdo i = 1, 4
   processcall mp_program(X(:,i), Y) location(p(i))
endprocessdo
```

As illustrated in Fig. 7, this program invokes an MP program (`mp_program`) on four virtual processors, with argument `X` partitioned and `Y` replicated. In the figure, the enclosing oval represents the FM process `execute_mp` while the execution graph represents the encapsulated MP computation. For brevity, we do not show the code required to set up the channels used for message-passing communication; however, this is simple, and can be incorporated automatically using a source transformation.

The same basic concepts can be used to integrate HPF into the multiparadigm framework. This integration has been demonstrated in a prototype compilation system developed with Bhaven Avalani and Alok Choudhary of Syracuse University [4]. HPF procedures are compiled to message-passing code using an HPF compiler; the message-passing code is then linked with both FM coordination code and interface routines that handle data transfer between the two languages. This system has been used to implement a variety of pipeline algorithms similar to those referred to earlier.

In summary, the techniques outlined in this section allow FM to be used as a coordination framework. This in turn allows us to construct multidisciplinary applications such as those illustrated in Figs. 1 and 6, with component models implemented using HPF or a message-passing library such as p4, Express, or MPI.

4.2 Experiences

We have developed an FM compiler that generates code for a variety of parallel computer platforms, including IBM SP, Intel Paragon, Cray T3D, Cray C90, and Ethernet- and ATM-connected workstations. Heterogeneous collections of these machines are also supported. Whenever possible, the compiler uses lightweight threads to implement FM processes, so as to reduce the cost of switching between multiple FM processes executing on the same processor. Performance studies show that communication performance is generally competitive with low-level message-passing libraries.

Programmers have used FM's task-parallel constructs for each of the three reasons discussed in the preceding section: software engineering, locality, and scheduling. In the software engineering area, an air quality model developed by Donald Dabdub and Rajit Manohar at Caltech is constructed from separate atmospheric transport and chemistry components. In another project, the ADIFOR automatic differentiation system generates FM code to exploit parallelism in derivative calculations. Task parallelism is used for locality in various pipelined task/data-parallel codes developed at Syracuse University. A parallel Fock matrix construction code uses the structure illustrated in Fig. 4 to achieve data-driven scheduling of computation and communication.

Several projects have used FM mechanisms to develop reusable libraries. For example, a Fock matrix program encapsulates concurrency and communication in a portable "global array" library, derived from a code originally developed by Robert Harrison using a nonportable interrupt-driven receive on Intel computers. This library is used in an ab initio quantum chemistry code which also incorporates a matrix diagonalization library integrated using the message-passing compatibility library. At Caltech, Dan Meiron and his colleagues have developed reusable templates for spectral computations.

5 Discussion

FM represents a conservative approach to task parallelism: its basic constructs are chosen to be simple and consistent with F77 concepts. Nevertheless, these concepts introduce powerful capabilities: dynamic process and communication structures; support for data-driven computation; programmer-control of resource allocation; compile-time guarantees of deterministic execution; and the compositionality required for library development. In this section, we examine various aspects of the language and contrast them with some alternative approaches. We also discuss the implications of this work for high-performance languages.

5.1 Explicit Parallelism

FM provides an explicitly parallel programming model. That is, its extensions have semantic content, and programs cannot easily be executed using a single thread of control. In other approaches, task parallelism is introduced via directives that, as in HPF, have no semantic content. If used correctly, a program compiled with and without directives will compute the same result. The directives serve merely to provide additional information to the compiler which helps it identify opportunities for parallel execution.

For example, the Fx Fortran compiler supports directives which can be used to specify pipeline algorithms [10]. The directives allow the programmer to supply data dependency information for iterative programs which apply a sequence of transformations to a stream of input data; the compiler uses this information to convert the program into a pipeline that streams data through data-parallel tasks responsible for performing the various transformations. Performance models are used to determine whether to generate pure data-parallel or mixed task/data-parallel code. The approach is illustrated in Fig. 8. The code on the left is a sequential program that performs a sequence of `m` two-dimensional

```
                                  C$     begin parallel
    do i=1,m                             do i=1,m
      call cffts(A)                        call cffts(A)
                       ===>       C$       output A
      call rffts(A)                        call rffts(A)
                                  C$       input A
    enddo                                enddo
                                  C$     end parallel
```

Figure 8: Using directives to specify pipeline algorithms in Fx Fortran

FFTs; the code on the right incorporates the directives needed for pipelined execution. The directives `begin parallel` and `end parallel` delineate the parallel loop; the directives `output` and `input` indicate that the array `A` is produced by `cffts` and consumed by `rffts`. Additional HPF-like directives in the FFT routines themselves specify data distributions.

This example illustrates two important advantages of directive-based approaches: existing sequential code need not be rewritten, and a compiler can tailor the generated code to problem size and machine characteristics. Disadvantages include the restricted set of applications that can be handled using any one directive, the complex compilers needed to exploit the information provided by directives, and the large semantic gap between the application program and the generated code.

We believe that while directives-based approaches are important, they do not avoid the need for explicit task parallelism. In our view, there will always remain both software engineering motivations for explicit task parallelism and problems that cannot be handled using directive-based approaches. In addition, even programs that can be compiled automatically must eventually be translated into executable task-parallel code. In both cases, it is useful to have an architecture-independent parallel notation, whether for use by the programmer or the compiler.

5.2 Determinism

While an explicitly parallel program may not have a straightforward sequential reading, it can still be possible to guarantee that it is deterministic: that is, that every execution of that program with a given input will produce the same output. In the vast majority of cases, this is the desired behavior: few parallel algorithms in science and engineering are nondeterministic, and indeed unwanted nondeterminism ("race conditions") in parallel programs has historically been a major source of problems [9].

Motivated by these observations, we chose when designing FM to make determinism the default behavior. To this end, we defined restrictive semantics for communication operations that ensure, for example, that each channel always has a single writer and a single reader. (The restrictions also have the advantage of simplifying implementation.) Nondeterminism is also supported, but must be introduced explicitly using specialized constructs.

Other approaches to task parallelism in Fortran do not enforce determinism. For example, in the shared-memory extensions proposed by ANSI committee X3H5 [1], programs can create explicit threads, which execute concurrently and interact by reading and writing shared data structures. It is the programmer's responsibility to prevent unwanted race conditions by using explicit locks or other mutual exclusion constructs to control access to shared data. Chapman et al. [3] have proposed mechanisms based on "spawn" and "shared data abstraction" constructs. The shared data abstraction, a form of monitor, is used to control interactions between tasks created using spawn. Again, the programmer must use these constructs in a structured fashion to ensure deterministic execution.

We believe that determinism is fundamental to parallel programming and that a compiler or runtime system must either prevent unwanted nondeterministic execution or be able to warn the programmer when it occurs. The FM approach to this problem appears to work well, although in some cases we fear that it is too draconian: in more complex nondeterministic executions (e.g., those involving the dynamic scheduling of a variable number of tasks) the FM restrictions can lead to convoluted code. We are investigating ways of relaxing these restrictions while maintaining the ability to enforce determinism when it is required.

5.3 Task Parallelism and HPF

We have described how FM can be used as a task-parallel coordination language for data-parallel HPF computations. This approach has proved quite effective, particularly for multidisciplinary and pipeline problems. However, we do not believe that it represents a satisfactory long-term solution to the problem of task parallelism in high-performance languages. Instead, we argue for a single language that integrates task- and data-parallel constructs.

A single language is preferable for several reasons. First, programming tools such as compilers, debuggers, and performance analyzers are more effective when they have access to information about all aspects of program behavior. A partitioning of computations into separate task-parallel and data-parallel worlds makes this more difficult. Second, many computations appear to require a fairly close intermingling of task and data parallelism. Again, this is hindered by the use of separate languages. Finally, we suspect that programmers are happier with an integrated programming model.

If we accept that task-parallel constructs are required in HPF, then FM suggests at least two directions that can be pursued in their design. First, we can build on FM *syntax* and introduce constructs that would allow an HPF computation to acquire processors, to initiate computation on these processors, and to connect computations on different sets of processors with virtual files on which read and write (or send and receive) operations could be performed. Notice that as these virtual file operations involve HPF processes, and each HPF process may execute on multiple processors, the communication logic required to implement even simple "write" and "read" operations can be complex.

Alternatively, we can build on FM *concepts* and define HPF extensions that support explicit task parallelism, modularity, determinism, and library construction, but using F90 rather than F77 constructs. For example, F90's dynamic, recursively-defined data structures provide a base on which we can build communication and synchronization mechanisms more flexible than the restrictive (but F77-like) channel construct of FM.

These mechanisms can then be used to implement libraries implementing virtual files or other higher-level interaction mechanisms. One candidate set of mechanisms uses single assignment variables for synchronization and remote procedure calls for communication [2].

Runtime and compiler design issues must also be addressed. An integrated task/data-parallel language may create concurrent task- and data-parallel computations on the same or different processors. The programmer, compiler, and runtime system need to cooperate to ensure efficient scheduling of these different computations. Important issues include mapping of computations to processors, coscheduling of threads of control belonging to the same data-parallel computation, and efficient organization of collective communication operations.

Acknowledgments

I am grateful to Mani Chandy, Alok Choudhary, Carl Kesselman, and Rob Schreiber for discussions on these topics. This work was supported by the National Science Foundation's Center for Research in Parallel Computation under Contract CCR-8809615 and by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] ANSI Technical Committee X3H5, *Parallel Processing Model for High Level Programming Models*, 1992.

[2] K. M. Chandy and C. Kesselman, CC++: A declarative concurrent object-oriented programming notation, *Research Directions in Concurrent Object-Oriented Programming*, Gul Agha, Peter Wegner, and Akinori Yonezawa (eds.), MIT Press, 1993.

[3] B. Chapman, P. Mehrotra, J. van Rosendale, and H. Zima, A software architecture for multidisciplinary applications: Integrating task and data parallelism, Technical Report 94-18, ICASE, MS 132C, NASA Langley Research Center, Hampton, Va., 1994.

[4] I. Foster, B. Avalani, A. Choudhary, and M. Xu, A compilation system that integrates High Performance Fortran and Fortran M, *Proc. 1994 Scalable High Performance Computing Conf.*, IEEE, 1994.

[5] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming, *J. Parallel and Distributed Computing*, 1994.

[6] D. Gannon et al., Implementing a parallel C++ runtime system for scalable parallel systems, *Proc. Supercomputing '93*, IEEE, 1993.

[7] P. Hatcher and M. Quinn, *Data-Parallel Programming on MIMD Computers*, MIT Press, 1991.

[8] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel, *The High Performance Fortran Handbook*, MIT Press, 1994.

[9] C. Pancake and D. Bergmark, Do parallel languages respond to the needs of scientific programmers?, *Computer* 23(12), 13–23, 1990.

[10] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross, Exploiting task and data parallelism on a multicomputer, *Proc. 4th ACM SIGPLAN PPoPP*, ACM, 1993.

Sidebar: Requirements

A workshop on "Task Parallelism in Fortran" held in Pasadena, February 9–10 1994, and sponsored by the NSF Center for Research on Parallel Computation, brought together participants from industry, universities, and federal laboratories to discuss requirements for task parallelism in Fortran. The workshop was intended not to discuss standards but to share information and to provide input for the second round of the HPF Forum. A detailed report is in preparation and will be published both electronically and in the journal *Scientific Programming.*

Participants in the workshop exchanged information regarding a wide range of approaches, including shared-memory extensions (SVM Fortran), directives (Fortran D, Fx Fortran, CHAOS) and explicit task-parallelism (Fortran M, Vienna Fortran, Large-Grain Dataflow). There was considerable consensus on requirements for a task-parallel Fortran, if not on actual mechanisms:

1. *Thread creation.* Most participants believed that dynamic thread creation is needed, although the SPMD model had some adherents. Both unstructured "spawn" and the more structured "parbegin/parend" constructs were considered useful.

2. *User-level resource management.* Most participants believed that programmers require control over how computational resources (processors) are allocated to computations, as well as the ability to specify mapping with respect to virtual rather than physical resources.

3. *Name spaces.* All agreed that some form of hierarchical name space is required, so that subcomputations can define and use local names.

4. *Thread interaction.* This was the area in which there was least consensus. Clearly, threads must be able to exchange data; however, there was no agreement whether this should be achieved by a shared address space with locks, channels, monitors, or other mechanisms. Many present felt that determinism was important.

Sidebar: For More Information

A World Wide Web/Mosaic information server at Argonne National Laboratory provides pointers to a range of research projects in the area of task and data parallelism in Fortran, and a database of applications requiring task and data parallelism. Its URL is `http://www.mcs.anl.gov/tpf`.

More information about the Fortran M language described in this article can be obtained via WWW/Mosaic at URL `http://www.mcs.anl.gov/fortran-m`. A Fortran M compiler and on-line documentation can also be obtained via anonymous ftp from `info.mcs.anl.gov`, in directory `pub/fortran-m`.

Sidebar: Languages or Libraries?

Parallelism can be either a second-class or first-class citizen in a programming language. That is, it can be incorporated via either libraries or language extensions. Each approach has its adherents.

In library-based approachs, parallelism is supported not in the language but in separate libraries that invoke machine-specific mechanisms to create threads of control, communicate, synchronize, etc. Programmers call functions defined by these libraries to create threads and to manage their execution. Numerous such libraries have been developed over the years, providing mechanisms for both shared-memory and distributed-memory computers (monitors, message passing, etc.), often in a portable fashion. The Argonne macros, p4, PVM, Linda, and MPI are just five examples. Proponents of libraries emphasize their simplicity and low cost: a new parallel library can be developed without the need to develop compiler technology, and the issue of standards is less critical.

Language-based approaches provide explicit language constructs for specifying parallel computation; these are translated by a compiler into appropriate low-level operations. Hence, the programmer uses a parallel block or a spawn statement to create a thread, communication operations or remote procedure calls to transfer data, and so on. Proponents of language extensions argue that programs are clearer when parallelism is specified using explicitly parallel constructs, and point to the benefits of compiler-based error detection and optimization. They also argue that the apparent simplicity of library-based approaches is misleading: the complexity associated with parallel programming does not disappear when a library is used, but is pushed up a level to be dealt with by the programmer using the library.

To some extent this controversy relates to the question of whether concurrency is regarded as a fundamental or incidental aspect of programming. In sequential programming, data structures and control structures tend to be viewed as fundamental and supported in languages, while incidentals such as input/output are relegated to libraries. Believing that task parallelism is fundamental rather than incidental, we focus on language extensions rather than libraries in this paper.