A Compilation System That Integrates High Performance Fortran and Fortran M

Ian Foster
Ming Xu

Mathematics and Computer Science
Argonne National Laboratory
Argonne, IL 60439

Bhaven Avalani
Alok Choudhary

CIS/CSE
Syracuse University
Syracuse, NY 13244

Abstract

Task parallelism and data parallelism are often seen as mutually exclusive approaches to parallel programming. Yet there are important classes of application, for example in multidisciplinary simulation and command and control, that would benefit from an integration of the two approaches. In this paper, we describe a programming system that we are developing to explore this sort of integration. This system builds on previous work on task-parallel and data-parallel Fortran compilers to provide an environment in which the task-parallel language Fortran M can be used to coordinate data-parallel High Performance Fortran tasks. We use an image-processing problem to illustrate the issues that arise when building an integrated compilation system of this sort.

1 Introduction

In data-parallel programming, programs apply a sequence of operations identically to all or most elements of a large data structure; in task-parallel programming, programs consist of a set of (potentially dissimilar) parallel tasks that perform explicit communication and synchronization operations. Both paradigms are supported by a number of parallel languages; in this paper, we use High Performance Fortran (HPF) [9] and Fortran M (FM) [5] as prototypical examples.

While task and data parallelism are often considered mutually exclusive approaches to parallel programming, many applications can in fact benefit from both forms of parallelism. This is the case in so-called multidisciplinary simulation, in which a computer model of a complex physical system (such as an aircraft or the earth's climate) is constructed from models of system components. A parallel implementation of such a system requires the ability to coordinate the execution of dissimilar (and possibly data-parallel) simulations. Similarly, many image-processing applications can be structured as pipelines of data-parallel operations. However, the development of integrated task/data parallel programming systems introduces challenging technical issues in the areas of language design, compilation, and run-time system design.

To enable early experimentation with the integration of task and data parallelism, both from the programming and implementation viewpoints, we are developing a system that allows the use of FM to coordinate concurrent HPF computations. This system integrates software and compiler techniques developed in the HPF compiler project at Syracuse and the FM compiler project at Argonne. In our prototype system, HPF and FM procedures are clearly distinguished and are compiled with the HPF or FM compiler, respectively. Simple interface routines are used to coordinate the two programming models. We are currently investigating mechanisms for a tighter integration of the two systems, with the eventual goal of producing a single language and compiler. We expect this effort to provide input into the next round of the HPF Forum, when it considers task parallelism.

A number of technical issues must be addressed when developing an integrated task/data parallel programming system. These fall into two general areas. In the area of *compiler and run-time design*, a primary concern is the reconciliation of techniques developed previously for task- and data-parallel languages. This issue is the main focus of this paper. The *language*

[0] To appear in: *Proc. 1994 Scalable High Performance Computing Conf.*, IEEE Computer Science Press.

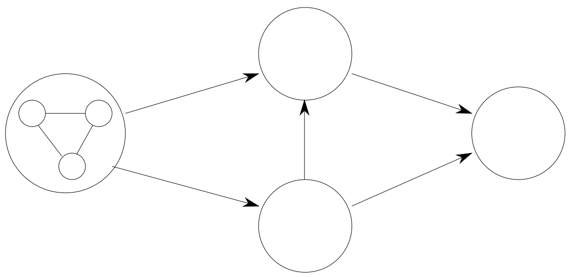

Figure 1: Fortran M Programming Model

design issues are concerned with how task- and data-parallel constructs are harmonized. In this paper, we adopt the approach proposed by Chandy et al. [2], namely, the use of distinct task- and data-parallel languages, with the task-parallel language used to coordinate data-parallel computations. An alternative approach that we will pursue in future work is the definition of a single integrated language.

The organization of this paper is as follows. Sections 2 and 3 introduce the FM and HPF compilers employed in the integrated compilation system. Sections 4 and 5 outline the techniques used to integrate FM and HPF and present a case study (an image-processing application) that benefits from the use of both task and data parallelism. Finally, Sections 6 and 7 review related work and present our conclusions.

2 Task parallelism

FM is a small set of extensions to Fortran for task-parallel programming. Programs are constructed by using single-reader, single-writer *channels* to plug together concurrent *processes*. Processes and channels can be created and deleted, and channels can be reconnected, dynamically; nevertheless, programs can be guaranteed to be deterministic. Virtual computer constructs can be used to control the mapping of processes to processors. This programming model is illustrated in Fig. 1, with processes represented by circles and channels represented by arrows.

An important feature of FM from the point of view of task/data-parallel integration is its support for compositionality. By this we mean that program components (processes) encapsulating concurrency, communication, and mapping decisions can be reused in different situations without concern for internal implementation details. In the HPF/FM system, FM processes are used to encapsulate data-parallel HPF computations, and virtual computers are used to control the allocation of computational resources to HPF computations.

The Argonne FM compiler compiles FM to Fortran 77 plus calls to a thread-management and message-passing library called Nexus [6]. Nexus message-passing functions are implemented in terms of TCP/IP, shared-memory operations, MPI, NX, or PVM, depending on the target architecture. Compilation proceeds without sophisticated analysis, with each FM statement being replaced with a mixture of Fortran code, calls to Nexus functions, and calls to compiler-generated interface routines, which in turn make calls to Nexus functions.

3 Data parallelism

HPF is a set of extensions to Fortran 90 that supports a data-parallel programming model. The Syracuse HPF compiler translates HPF programs into Fortran 77 plus calls to a set of collective communication library routines [1]. It exploits only the parallelism expressed in HPF's explicitly data-parallel constructs such as forall, array constructs, the independent do assertion, and intrinsic functions. It does not attempt to parallelize other constructs. The foundation of the compiler lies in recognizing commonly occurring communication and computation patterns. These patterns are then replaced by calls to the optimized run-time support system routines. This approach represents a significant departure from traditional parallelizing compilers, which perform in-depth dependency analysis to recognize parallelism and embed all synchronization and low-level communication functions inside the generated code.

The basic compiler is divided into three parts.

- Translator: This translates HPF to F77 plus calls to communication libraries.
- Communication Library: Express is currently used as the communication medium. A set of transport-independent routines is under development, which will run on communication systems such as MPI, PVM, and p4.
- Intrinsics: This is a set of routines acting on vector data elements. These include reduction routines, shift routines, and general mathematical routines such as matrix multiplication and matrix transpose.

The compiler works in the following phases to generate the required code.

- Data Partitioning: The compiler partitions the data and computation on the processors following the `decomposition`, `align`, and `distribute` directives given in the HPF program.

- Communication Detection and Generation: The compiler detects and generates communication using pattern matching to detect commonly occurring communication patterns and generating the equivalent communication calls to the communication libraries.

- Code Generation: The code generator produces loosely synchronous SPMD code. The generated code is structured as alternating phases of local computations and global communications. In such a model the processes do not need to synchronize during local computations.

4 Integration

We now discuss our approach to the integration of task and data parallelism.

4.1 Applications

Several broad classes of mixed task/data-parallel problems can be distinguished. Probably most common are those involving task-parallel collections of data-parallel subcomputations. In the simplest case, the set of data-parallel computations is static, and hence the task-parallel component of the program is concerned only with initiating the data-parallel computations and coordinating the flow of data between these computations. Problems of this sort include some multidisciplinary simulations (e.g., a coupled climate models with ocean and atmosphere components) and simple image-processing pipelines. In more complex cases, the network of data-parallel computations may change during program execution. This is the case in more complex multidisciplinary simulations, in image-processing problems where the computation depends on the image, and in search applications where the evaluation function applied at each search tree node is a data-parallel program.

In another class of problem, a data-parallel program calls task-parallel programs. This approach may be required if a data-parallel computation includes a phase that is not data parallel: a call to a task-parallel routine can avoid a need for sequential execution. Alternatively, a data-parallel computation may require that different task-parallel computations be performed on different data items. For example, a function applied in data-parallel fashion to each component of a data structure may need to initiate a task-parallel function evaluation on some components.

4.2 Our approach

In this paper, we consider only the relatively simple situation in which distinct task- and data-parallel languages are defined, and the task-parallel language is used to coordinate data-parallel computations. In particular, we assume that FM is used to create networks of data-parallel HPF computations, which may themselves interact by sending and receiving messages on channels established by the FM coordinating program.

An FM program that calls a data-parallel routine must be able to indicate the HPF routine that is to be executed and the number and identity of the processors in which it is to execute. Invocation of an HPF computation is indicated by a special `HPFCALL` statement, which specifies the name of the routine to execute and its arguments. For example:

```
HPFCALL my_hpf_proc(a1,...,an)
```

Arguments passed from FM to HPF can be either replicated or distributed over the HPF virtual processors [7]. The number of processors is determined by the FM virtual computer in which the call is made.

An HPF program passed a port (or, more commonly, an array of ports) as an argument can communicate with other, concurrently-executing, HPF computations by using HPF's extrinsic procedure mechanism to invoke FM communication routines. Alternatively, and more elegantly, we can extend HPF with port data types and data-parallel send and receive operations, allowing us to write code such as the following:

```
      processors pr(8)
      real A(M,N)
      outport (real x(M,N)) outp(8)
$HPF  DISTRIBUTE A(BLOCK,*)
      send(outp) A
```

where `outp` is, as shown, an array of FM ports. We assume the latter approach in this paper.

4.3 Technical issues

Technical issues that must be resolved to implement this approach include the following.

Invocation. The **HPFCALL** statement can be implemented in terms of an existing syntactic extension, **MPCALL**. This statement, which allows invocation of arbitrary message-passing programs, is implemented via a preprocessor that generates an interface routine that invokes the HPF program [7].

Data Sharing. An FM program can pass FM data structures as arguments to an HPF procedure; modified versions of these data structures may be returned upon completion of the HPF computation. As FM and HPF may use different data representations and distributions, data conversion may be required at the FM/HPF interface. We incorporate this conversion in the interface routines generated by the **MPCALL** preprocessor. Currently, the conversion is simple because the Syracuse HPF uses simple data layouts. In the future, it could be more complex [2].

Data-Parallel Channel Communication. Our compilation system currently assumes that two communicating HPF computations execute on the same number of (virtual) processors and use different decompositions for the variables that they communicate. In the general case, however, processors may use data distributions, in which case data redistributions are required as part of communication. This situation is similar to the parallel I/O problem (where in-memory and disk files may be laid out in different ways), and similar solutions can probably be employed [11].

Runtime Systems. The code generated by the HPF and FM compilers includes calls to the HPF and FM run-time systems, which provide collective communication and task/message management routines, respectively. The two run-time systems cannot coexist directly, because the HPF system makes the natural assumption that the generated code is to run independently of other computations. In particular, it does not allow for multiple independent HPF computations running on the same processor, or for a HPF computation using a subset of available processors.

The two run-time systems can be harmonized either by integration or by layering. Integration involves the development of a common run-time system providing facilities required by both languages; layering implements HPF run-time system functions in terms of FM or Nexus facilities. We have adopted the latter approach and have developed a message-passing compatibility library written in FM that implements the Express calls generated by the HPF compiler as operations on channels set up when the HPF code is invoked from FM [7].

The merit of this layered approach is that it requires no changes to either run-time system. A disadvantage is that it hinders the use of optimized collective communication algorithms within data-parallel components. In principle, these algorithms could be invoked by an optimizing FM compiler; in the absence of such a compiler, however, all communication is performed using generic FM mechanisms. The development of integration techniques that preserve the performance of data-parallel components will be a topic of future research.

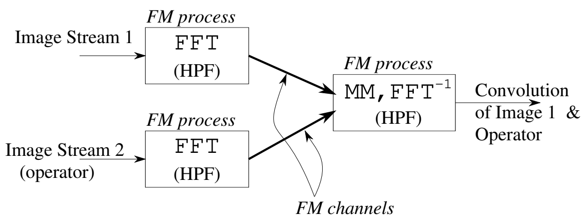

Figure 2: Convolution Algorithm Structure

Heterogeneous Environments. A task/data parallel program is inherently heterogeneous in that different parts of the computation may execute different code. This heterogeneous program structure may then be mapped to a heterogeneous computer system, either for reasons of convenience or because some components of the application can benefit from specialized computer resources. This complicates both mapping and run-time optimization. As the FM compiler is designed to execute in heterogeneous environments, our compilation system will allow us to explore these issues in future work.

5 2-D convolution: A case study

Two-dimensional (2-D) convolution is a simple example of a problem that can benefit from both task and data parallelism. We use it here to illustrate the mechanisms used to integrate FM and HPF. As shown in Fig. 2, 2-D convolution involves two 2-D fast Fourier transforms (FFT), an element-wise multiplication, and an inverse 2-D FFT.

5.1 Formulation

A data-parallel convolution algorithm can be formulated in HPF as in Fig. 3. Images are computed one at a time, with data distribution statements used to indicate how data parallelism is to be exploited within each FFT. The 2-D FFT uses the Fortran 90 `transpose` intrinsic to transpose the input array so that 1-D FFTs can be performed in each dimension without communication. Each FFT operates on all processors.

A mixed data/task-parallel algorithm can use disjoint subsets of processors for each FFT. Data flows between components in a pipeline fashion, which can improve overall performance by reducing the number of communications required. The mixed algorithm can be formulated in HPF/FM as illustrated in Fig. 4. (For brevity, the FM code uses some Fortran 90 constructs as abbreviations; these are not yet supported by the FM compiler.) This comprises a main program, written in FM, and two HPF wrapper routines, `fftW` and `iftW` (the latter is not shown), which invoke the corresponding HPF FFT routines. The main program creates the structure illustrated in Fig. 2. The `HPFCALL` statements create the boxes in the figure and the channels the connections between the boxes. For simplicity, input and output are not considered. The `SUBMACHINE` annotations control resource allocation.

The HPF wrapper routine `fftW` repeatedly initializes data, invokes the HPF FFT routine, and forwards output data. Because this is data-parallel code, the `SEND` statement represents a parallel send: the communication of the distributed array `A` on an array of ports.

In summary, the mixed code comprises three components: the FM main program, the HPF wrapper routine `fftW`, and the HPF two-dimensional `fft` of Fig. 3.

5.2 Implementation

Eventually, we expect to provide a single compiler that will accept both FM and HPF constructs. This will allow the mixed-algorithm code to be compiled and executed directly. Currently, this capability is not provided, and the generation of an executable program requires some additional programmer intervention. Such intervention is not required for the FM main program or for the HPF FFT routine: these can be compiled using the FM and HPF compilers, respectively, to produce a Fortran 77 `main` and `f77fft` subroutine.

Because the HPF compiler does not yet recognize communication constructs, we must hand-compile the interface routine `fftW` to generate the following FM code. Note that this is the code executed on a single virtual processor. It receives input data, invokes the FFT code generated by the HPF compiler when applied to the HPF routine `fft` (`f77fft`), and forwards output data.

```
C       HPF Code
        program data_parallel
        complex A(512,512), B(512,512)
        complex C(512,512)
!HPF$   processors pr(24)
!HPF$   align B, C with A
!HPF$   distribute A(BLOCK,*)
        do i = 1,nimages
          call read_input(A,B)
          call fft(512,512,A)
          call fft(512,512,B)
          C = A*B
          forall(i=1:512, j=1:512)
              C(i,j) = CONJG(C(i,j))
          call ifft(C)
          call write_output(C)
        enddo
        end


C       Two-dimensional forward FFT
        subroutine fft(nrows,ncols,A)
!HPF$   processors pr(24)
        complex A(nrows,ncols)
!HPF$   distribute A(:,BLOCK)
        call d1fft(nrows,ncols,A)
        A = transpose(A)
        call d1fft(nrows,ncols,A)
        end


C       One-dimensional FFT on 2-d array
        subroutine d1fft(nrows,ncols,A)
        complex A(nrows,ncols)
!HPF$   processors pr(24)
!HPF$   distribute A(:,BLOCK)
!HPF$   independent
        do i = 1,nrows
          call rowfft(i,A)
        enddo
        end

        function rowfft(icol,A)
!HPF$   pure
        ...
        end
```

Figure 3: Data-Parallel Convolution Program

```
C       FM Code
        process fft(nimages,out)
        outport (complex x(512,512/8)) out
        complex Alocal(512,512/8)
        do i = 1,nimages
          call read_input(Alocal)
          call f77fft(Alocal)
          send(out) Alocal
        enddo
```

This code is then compiled with the FM compiler and linked with `main`, `f77fft`, and various FM libraries to produce an executable program.

```
C       FM Code
        program mixed_parallel
        processors pr(24)
        outport (complex x(512,64)) os1(8)
        outport (complex x(512,64)) os2(8)
        inport (complex x(512,64)) is1(8)
        inport (complex x(512,64)) is2(8)
        channel(in=is1(:), out=os1(:))
        channel(in=is2(:), out=os2(:))
        ...
        processes
         hpfcall fftW(img,os1) submachine(1:8)
         hpfcall fftW(img,os2) submachine(9:16)
         hpfcall iftW(is1,is2) submachine(17:24)
        endprocesses
        end


C       HPF Code
        subroutine fftW(nimages, outs)
        outport (complex x(512,512)) outs(8)
        complex A(512,512)
!HPF$   processors pr(8)
!HPF$   distribute A(:,BLOCK)
        do i = 1,nimages
         call read_input(A)
         call fft(A)
         send(outs) A
        enddo
```

Figure 4: Task/Data-Parallel Convolution Program

5.3 Performance

We give preliminary performance results for the data-parallel and mixed codes in Fig. 5. These results were obtained on a 128-processor IBM SP1 using TCP/IP for interprocessor communication. (The SP1 also provides a more efficient communication library called EUI-H. At the time when these experiments were performed, however, the FM compiler had not yet been modified to generate calls to EUI-H routines.) For these experiments, the FM main program and interface routines were compiled with the FM compiler, while the HPF code was hand-compiled. We see that the mixed code gives superior performance for smaller problems and on larger numbers of processors, while the pure data-parallel code is faster on larger problems and on smaller numbers of processors.

This result can be explained using a simple performance model. Assume that the convolution code is applied to arrays of size $N \times N = D$ on P processors and that the cost of sending a message of N words is $t_s + t_w N$, where t_s and t_w are constants. The data-parallel model performs three matrix transposes, each involving P processors. If P divides N, then each matrix transpose requires $P(P-1)$ messages and the communication of $D(P-1)/P$ data. Hence, total communication costs are

$$T_{\text{data}} = t_s 3P(P-1) + t_w 3D\frac{P-1}{P}.$$

The mixed algorithm also performs three data-parallel FFTs, each on $P/3$ processors. In addition, the $P/3$ processors responsible for each of the forward FFTs must communicate D data to the $P/3$ processors responsible for the inverse FFT. If $P/3$ divides N and $P \geq 3$, then total communication costs are

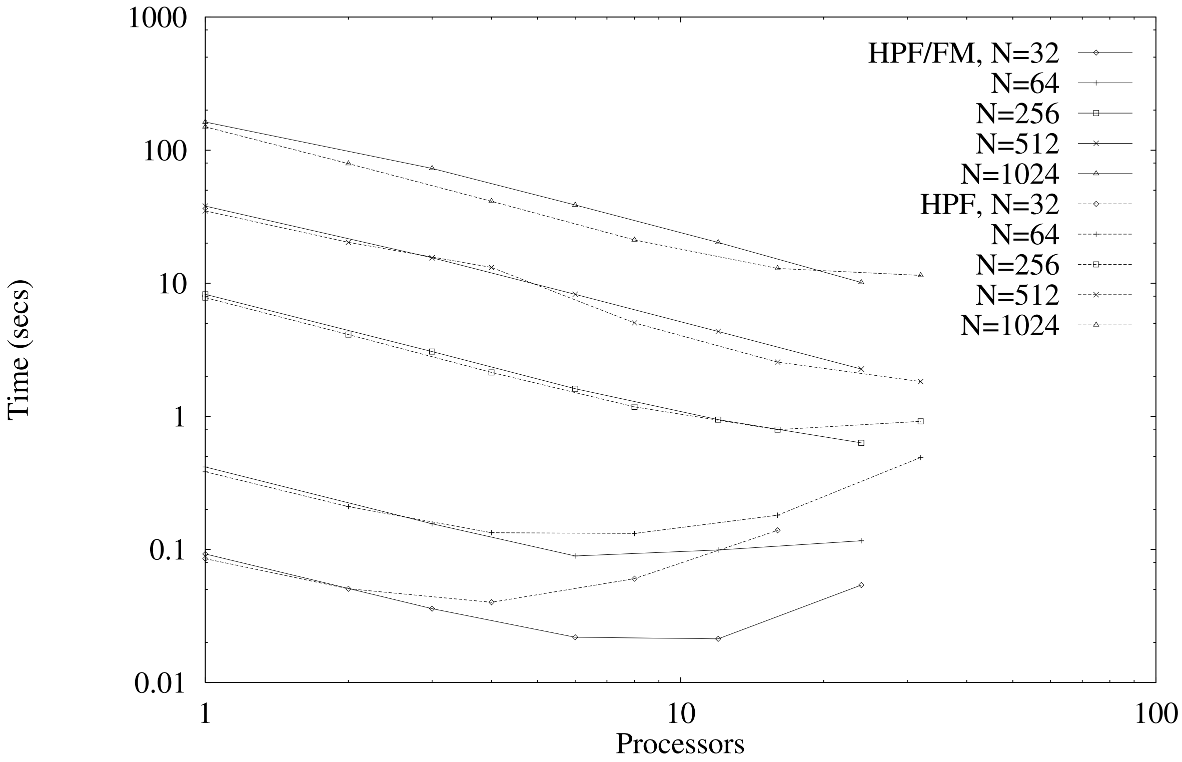

Figure 5: Convolution Code Performance on IBM SP1

$$T_{\text{mixed}} \quad = \quad t_s \left(P + \frac{P^2}{3} \right) + t_w D \left(2 + \frac{(P/3-1)}{P} \right)$$

In summary, the mixed algorithm sends fewer messages but more data. Hence, we can expect it to be faster when N is small and P is large, the situation that we in fact see in practice.

6 Related work

Multiparadigm programming has been investigated by many researchers in a sequential context (for example, see [13]). In a parallel context, most work has focused on task-parallel coordination frameworks for data-parallel components. For example, Cheng et al. propose the use of the AVS dataflow visualization system to implement multidisciplinary applications, in which some components may be data-parallel programs [4]. This provides an elegant graphical programming model but is less expressive than HPF/FM. For example, cyclic communication structures are not easily expressed. Similarly, Quinn et al. describe work on iWARP in which a configuration language is used to connect Dataparallel C computations [8]. The Dataparallel C programs use specialized versions of C I/O libraries for communication. Process and communication structures are static, and all communication passes via a central communication server.

Subhlok et al. describe a compile-time approach for exploiting task and data parallelism on the iWarp mesh-connected multicomputer [12]. The input program incorporates HPF-like data parallel constructs and directives indicating data dependencies and parallel sections. An appropriate mix of task and data parallelism is then generated automatically by the compiler. HPF/FM permits more general and dynamic forms of task parallelism but also requires more programmer intervention.

Chapman et al. [3] have recently proposed the addition of "spawn" and "shared data abstraction" constructs to data-parallel languages to support multidisciplinary programming. The shared data abstraction, a form of monitor, is used to control interactions between tasks created using spawn. A novel aspect of this proposal is that shared data abstractions are persistent entities. While there has not yet been any experience in using these mechanisms for building applications, it appears that they are intended primarily for relatively coarse-grained computations.

7 Conclusions

We have developed a prototype programming system that allows mixed task/data-parallel programs coded in FM and HPF to be compiled and executed on parallel computers. The system has been used to implement a simple image-processing application; experimental studies confirm the benefits of a mixed task/data-parallel formulation, in that the mixed algorithm achieves higher performance than a pure data-parallel program.

The development of this prototype system is just the first phase in a research program designed to investigate the language constructs, compiler techniques, and run-time mechanisms required to support efficient mixed paradigm parallel programs.

Acknowledgments

This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the National Science Foundation's Center for Research in Parallel Computation under Contract CCR-8809615; and by ARPA under Contract DABT63-91-C-0028. We are grateful to other members of the Center for Research on Parallel Computation's Paradigm Integration project, particularly Mani Chandy, Carl Kesselman, Robert Olson, and Steven Tuecke, for their contributions to this work.

References

[1] Z. Bozkus, Alok Choudhary, Geoffrey C. Fox, T. Haupt, and S. Ranka, Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance eesults. In *Proc. Supercomputing '93*, IEEE, November 1993, Portland, Oregon.

[2] K. M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng, Integrated support for task and data parallelism, *Intl. J. Supercomputer Applications*, 8(2), 1994 (in press).

[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 23681, 1994.

[4] G. Cheng, G. Fox, and K. Mills, Integrating multiple programming paradigms on Connection Machine CM5 in a dataflow-based software environment, Technical Report, NPAC, Syracuse University, Syracuse, N.Y., 1993.

[5] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *J. Parallel and Distributed Computing* (to appear), and Preprint MCS-P327-0992, Argonne National Laboratory, 1992.

[6] I. Foster, C. Kesselman, R. Olson, and S. Tuecke, Nexus: An Interoperability toolkit for parallel and distributed computer systems, Technical Report ANL/MCS-TM-189, Argonne National Laboratory, 1994.

[7] I. Foster and M. Xu. Libraries for parallel paradigm integration. In *Toward Teraflop Computing and New Grand Challenge Applications*, R. Kalia and P. Vashishta (eds.), Nova Science Publishers, 1994.

[8] P. Hatcher and M. Quinn. *Data-parallel Programming on MIMD Computers.* The MIT Press, Cambridge, Mass., 1991.

[9] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, Texas, January 1993.

[10] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *CACM*, 35(8):66–80, August 1992.

[11] J. del Rosario and A. Choudhary, High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, March 1994.

[12] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, Calif., May 1993.

[13] P. Zave, A compositional approach to multiparadigm programming, *IEEE Software*, 15–25, 1989.