To appear in: Intl. J. Supercomputer Applications

Integrated Support for Task and Data Parallelism*

Mani Chandy
Center for Research on Parallel Computation
California Institute of Technology
Pasadena, CA 91125

Ian Foster
Argonne National Laboratory
Argonne, IL 60439

Ken Kennedy
Charles Koelbel
Chau-Wen Tseng
Center for Research on Parallel Computation
Rice University
Houston, TX 77251-1892

August 27, 1993

Abstract

We present an overview of research at the CRPC designed to provide an efficient, portable programming model for scientific applications possessing both task and data parallelism. Fortran M programs exploit task parallelism by providing language extensions for user-defined process management and typed communication channels. A combination of compiler and run-time system support ensures modularity, safety, portability, and efficiency. Fortran D and High Performance Fortran programs exploit data parallelism by providing language extensions for user-defined data decomposition specifications, parallel loops, and parallel array operations. Compile-time analysis and optimization yield efficient, portable programs. We design an interface for using a task-parallel language to coordinate concurrent data-parallel computations. The interface permits concurrently executing data-parallel computations to interact through messages. A key notion underlying the proposed interface is the integration of Fortran M resource management constructs and Fortran D data decomposition constructs.

*This research was supported by the National Science Foundation's Center for Research in Parallel Computation, under Contract CCR-8809615.

1 Introduction

The primary goal of the Center for Research on Parallel Computation is to make parallel computers easier to use. We feel that one of the fundamental problems in the use of parallel computers today is the difficulty of writing parallel software. Part of this difficulty stems from the fact that no universally acceptable programming paradigm for parallel machines has been developed. Some programs are most naturally considered as sets of interacting, large-grain tasks; others are naturally thought of as many fine-grain operations performed on the elements of a large data structure; still other programs may have other natural expressions. Moreover, the same paradigm may be easy to implement on one architecture, but very difficult to implement efficiently on another. Recent reports have indicated that even single applications may require several paradigms in their implementation; see, for example, (Federal HPCC Program, 1993).

In this paper we approach this multiple-paradigm situation directly: We describe two state-of-the-art parallel programming paradigms for supporting task and data parallelism, show how they can be integrated, and provide a preliminary design for the necessary interfaces. Some of these ideas are being investigated in a compilation system under development at Argonne National Laboratory and Syracuse University (Foster et al., 1994). First, however, we define both task parallelism and data parallelism, then motivate their integration.

1.1 Task Parallelism

In a task-parallel programming paradigm the program consists of a set of (potentially dissimilar) parallel tasks that interact through explicit communication and synchronization. Task parallelism may be both synchronous and asynchronous. In this paper, we use Fortran M (FM) as an example of a language supporting such a programming paradigm (Foster and Chandy, 1992; Chandy and Foster, 1993). Section 2 describes the FM language in more detail. Here, we focus on the advantages and disadvantages of the task-parallel paradigm.

A major advantage of task parallelism is its flexibility. Because of its emphasis on explicit coordination of individual tasks (or processes, as they are often called), task parallelism can be used to exploit both structured and unstructured forms of parallelism. Many scientific applications contain task parallelism. For example, in a climate model application the atmospheric and ocean circulation may be computed in parallel as two separate tasks. A task-parallel language can express this relationship easily, even if different methods are used for the two circulation models. Another natural application of task-parallel languages is reactive systems in which tasks must produce output in response to changing inputs, in a time-dependent manner. Tasks may also be organized as a pipeline to exploit pipeline parallelism.

Since interactions between tasks are explicit, the programmer can write programs that exploit parallelism not detectable automatically by compiler techniques. The programmer may also carefully tune the application so that it includes only the communication and synchronization that is actually necessary or efficient, hence reducing reliance on compiler optimizations. In general, task parallelism is less dependent on advanced compiler technology than is data parallelism; in many cases, all that is strictly necessary is the translation of task interactions into appropriate low-level primitives on the target architecture. However, compiler technology is still important as a means of guaranteeing correct execution and permitting representations of communication and synchronization that are convenient for the programmer.

A disadvantage of the task-parallel programming model is that it requires extra effort from the programmer to create explicit parallel tasks and manage their communication and synchronization. It is also often convenient to consider data owned by different tasks as being part of a single data structure; many task-parallel languages do not support this view directly. Because communication and synchronization are explicit, changing the manner a program is parallelized may require extensive modifications to the program text.

1.2 Data Parallelism

In a data-parallel programming paradigm the program consists of a series of operations that are applied identically to all or most elements of a large data structure. Parallelism may be implicit or explicit, with additional annotations describing how the data structures are decomposed and distributed among the physical processors. Data parallelism may be both regular (e.g., dense matrices) or irregular (e.g., sparse matrices). In this paper, we use Fortran D (Fox et al., 1990) and High Performance Fortran (HPF) (High Performance Fortran Forum, 1993) as examples of languages that support a data-parallel programming paradigm. Section 3 describes these languages in more detail.

A major advantage of data parallelism derives from its scalability. Because operations may be applied identically to many data items in parallel, the amount of parallelism is dictated by the problem size. Higher amounts of parallelism may be exploited by simply solving larger problems with greater amounts of computation. Data parallelism is also simple and easy to exploit. Because data parallelism is highly uniform, it can usually be automatically detected by an advanced compiler, without forcing the user to manage explicitly processes, communication, or synchronization.

Many scientific applications may be naturally specified in a data-parallel manner. Operations on whole data structures, such as adding two arrays or taking the inner product of two vectors, are common, as are grid-based methods for solving partial differential equations (PDEs). Since data-parallel programs are relatively close to sequential programs, many compiler analysis and optimization techniques can be adapted to produce parallel programs automatically. The mapping of data and computation can affect performance significantly (Knobe, Lukas, and Steele, 1990). With data-parallel programs, relatively simple data decomposition annotations are sufficient to achieve high performance on advanced parallel architectures. If communication and parallelism are implicit (as in HPF), the user can easily tune the program by small modifications to its data decomposition annotations.

A disadvantage of the data-parallel programming paradigm is that it is less general than task parallelism. Data-parallel programs may be converted into task-parallel programs by investing programmer effort, but the reverse is difficult to achieve with reasonable efficiency. Applications that require heterogeneous processing are at best difficult to express in data-parallel languages. Also, despite promising early tests it is not known how well compilers can optimize data-parallel programs.

1.3 Integrating Parallel Paradigms

As the discussions above make clear, task parallelism and data parallelism have complementary strengths that are appropriate for different problems. Perhaps less obvious is the fact that both paradigms may be needed in the same application. An excellent example of this is a multidisciplinary simulation. Simulating high-performance aircraft may involve many models, including fluid dynamics, structural mechanics, surface heating, and controls. Some formulations of such applications allow several disciplines to be solved simultaneously via task parallelism. Within each discipline the computations often have substantial data parallelism; this is certainly the case for the fluid dynamics, structural mechanics and surface heating models in our example. Other examples of mixed task and data parallelism include (Federal HPCC Program, 1993)

- environmental models: atmospheric dynamics, surface water, and ground water models;
- predictive cellular organelle models: fluid dynamics, rigid body mechanics, molecular dynamics, and other models; and
- multilevel models in a single discipline: linear, Euler, and Navier-Stokes models in aerodynamics.

Many Grand Challenge computations depend on such multidisciplinary approaches.

Applying only one parallel paradigm to the applications listed above may result in lower performance than an integrated approach might achieve. If only the task-parallel level is used, then only a limited amount of parallelism may be available. In our multidisciplinary aircraft simulation example, there are only four tasks at this level; running such a program on a 128-processor hypercube would waste most of the available capacity.

Attacking only the data-parallel level may cause other inefficiencies. Different disciplines may have different levels of parallelism, making it difficult to map them all to the same machine. Moreover, communication and synchronization overheads grow with machine size; at some point, it is not profitable to apply more processors to a single data-parallel computation.

Experience suggests that the best performance can often be obtained by combining approaches. Each high-level task can be a data-parallel computation. The task-parallel language can be used to allocate machine resources to balance the computational load and coordinate between these tasks. Within individual tasks, a data-parallel language can be used to take advantage of the parallelism within the discipline. Proper balancing of the high-level tasks and efficient computation within each data-parallel task can then ensure that the machine is efficiently used.

The situation above, although common, is not the only possibility for combining the two paradigms. It may also be desirable for a data-parallel program to call a task-parallel language. For example, all but one phase of an application may be naturally data parallel. Using task parallelism on the remaining phase (assuming it is appropriate) then removes a sequential bottleneck from the overall program.

The remainder of this paper describes programming languages and techniques for supporting the task-parallel and data-parallel programming models and technical requirements for their integration. Sections 2 and 3 describe the relevant features of the FM and Fortran D/HPF languages. Section 4 describes the design of the interface and some implementation issues. Section 5 describes related work, and Section 6 gives our conclusions and plans for future work.

2 Fortran M

Fortran M (FM) is a language designed by researchers at Argonne and Caltech for expressing task-parallel computation (Foster and Chandy, 1992). It comprises a small set of extensions to Fortran and provides a message-passing parallel programming model, in which programs create processes that interact by sending and receiving messages on typed channels. Two key features of the extensions are their support for determinism and modularity.

A prototype FM compiler has been developed and used to build libraries of parallel programs in linear algebra, spectral methods, mesh computations, computational chemistry, and computational biology. The compiler is available by anonymous ftp from `info.mcs.anl.gov`, in directory `pub/pcn`.

FM is currently defined as extensions to Fortran 77. However, equivalent extensions can easily be defined for Fortran 90 (American National Standards Institute, 1990), and we plan to extend the FM compiler to accept Fortran 90 syntax. For clarity, we use some Fortran 90 syntax in subsequent sections when discussing integration of FM and HPF.

2.1 Concurrency and Communication

FM provides constructs for defining program modules called *processes*; for specifying that processes are to execute concurrently; for establishing typed, one-to-one communication *channels* between processes; and for sending and receiving messages on channels. Send and receive operations are modeled on Fortran file I/O statements, but operate on *port variables* rather than unit numbers.

The FM programming model is dynamic: processes and channels can be created and deleted dynamically, and references to channels can be included in messages. Nevertheless, computation can be guaranteed to be deterministic; this feature avoids the race conditions that plague many parallel programming systems (Pancake and Bergmark, 1990). Determinism is guaranteed by defining operations on port variables to prevent multiple processes from sending concurrently, by requiring receivers to block until data is available, and by enforcing a copy-in/copy-out semantics on variables passed as arguments to processes. Nondeterministic constructs are also provided for programs that operate in nondeterministic environments.

Figure 1 illustrates the use of several FM constructs. The first code fragment uses the `CHANNEL` statement to create two channels and a process block (delineated by `PROCESSES` and `ENDPROCESSES` statements) to create two processes called `CONTROLS` and `DYNAMICS`. 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 a FM compiler to generate efficient communication code and to convert to a machine-independent format in a heterogeneous environment.

```
PROGRAM AERODYNAMICS
INPORT (INTEGER, REAL X(10,20), REAL Y(10,20)) PI
OUTPORT (INTEGER, REAL X(10,20), REAL Y(10,20)) PO
...
CHANNEL(IN=PI,OUT=PO)
CHANNEL(IN=QI,OUT=QO)
...
PROCESSES
   PROCESSCALL CONTROLS(PI,QO)
   PROCESSCALL DYNAMICS(QI,PO)
ENDPROCESSES
END



PROCESS CONTROLS(IN,OUT)
INPORT (INTEGER, REAL X(10,20), REAL Y(10,20)) IN
OUTPORT (INTEGER, INTEGER, REAL X(10,10,3)) OUT
...
SEND(OUT) I, J, A
RECEIVE(IN) NSTEP, U, V
...
END
```

Figure 1 Example FM Program

2.2 Resource Management

FM resource management constructs allow the programmer to specify how processes and data are to be mapped to processors and hence how computational resources are to be allocated to different parts of a program. These constructs influence performance but not correctness. Hence, we can develop a program on a uniprocessor and then tune performance on a parallel computer by changing mapping constructs.

FM process placement constructs are based on the concept of a virtual computer: a collection of virtual processors, which may or may not have the same shape as the physical computer on which a program executes. A virtual computer is an N-dimensional array, and mapping constructs are modeled on array manipulation constructs. The `PROCESSORS` declaration specifies the shape and dimension of a processor array, the `LOCATION` annotation maps processes to specified elements of this array and, as in PCN, the `SUBMACHINE` annotation specifies that a process should execute in a subset of the array (Foster, Olson, and Tuecke, 1992). For example, the following code places the `CONTROLS` and `DYNAMICS` processes on different virtual processors.

```
PROCESSORS(2)
...
```

```
PROCESSES
  PROCESSCALL CONTROLS(...) LOCATION(1)
  PROCESSCALL DYNAMICS(...) LOCATION(2)
ENDPROCESSES
```

In contrast, the following code places each process in a submachine comprising 10 virtual processors. This would be useful, for example, if the processes were themselves parallel programs, written in FM or HPF.

```
PROCESSORS(20)
...
PROCESSES
  PROCESSCALL CONTROLS(...) SUBMACHINE(1:10)
  PROCESSCALL DYNAMICS(...) SUBMACHINE(11:20)
ENDPROCESSES
```

2.3 Data Distribution

Although the basic paradigm underlying FM is *task parallelism*, the language also provides some support for data-parallel computation. Inspired by Fortran D and HPF, it permits programs to use data distribution statements (as discussed in the next section) to create distributed arrays (Chandy and Foster, 1993). (However, this capability is not supported in the prototype compiler.) Semantically, distributed arrays are indistinguishable from nondistributed arrays. That is, they are accessible only to the process in which they are declared and are copied when passed as arguments to subprocesses. Operationally, elements of a distributed array are distributed over the nodes of the virtual computer in which the process is executing. Hence, operations on a distributed array may require communication.

Data-distribution statements allow FM programs to declare arrays that are larger than can fit in a single processor's memory, and to specify certain limited classes of data-parallel computations on these arrays. However, their principal utility is as a mechanism for integrating task- and data-parallel computations. This application is discussed in detail below.

2.4 Implementation Issues

The Fortran subset of FM can be compiled with conventional compilers. Hence, the primary difficulty that arises in compiling FM is achieving efficient implementations of communication, synchronization, and process-management mechanisms. A prototype FM compiler has been developed at Argonne to facilitate both exploration of these issues and experimentation with task-parallel programming.

The prototype compiler is a preprocessor that translates FM programs into Fortran 77 plus calls to a runtime library called Nexus. Nexus provides basic services required to implement FM on parallel computers, such as multiple threads of control, global pointers, and remote procedure calls. Nexus services may be implemented using different technologies on different parallel computers: for example, portable message-passing libraries on heterogeneous networks, low-level message-passing or active messages on multicomputers, shared-memory operations on multiprocessors, or Windows NT services on networks of PCs.

The prototype compiler is currently operational on networks of workstations, the IBM SP-1 multicomputer, and the CRAY C90 (Foster, Olson, and Tuecke, 1993). It supports all language constructs except those concerned with data distribution. Preliminary experiments suggest that the performance of basic communication and synchronization operations is competitive with, and in some cases superior to, conventional message-passing libraries (Foster and Chandy, 1992).

3 Fortran D and High Performance Fortran

Fortran D was an early language designed by researchers at Rice and Syracuse for expressing data-parallel computation (Fox et al., 1990). It provided data decomposition specifications, parallel loops, and methods for specifying reductions. Fortran D was designed to support machine-independent parallel programming and was motivated by the observation that few languages support efficient data placement (Pancake and Bergmark, 1990). It brought together ideas found in earlier parallel languages such as CM Fortran (Thinking Machines Corporation 1991) and Kali (Koebel and Mehrotra, 1991), but in a unified framework designed to permit advanced compiler analysis and optimization.

Fortran D sparked widespread interest and strongly influenced the development of High Performance Fortran (HPF) (High Performance Fortran Forum, 1993), a set of extensions and modifications to Fortran 90 defined by a coalition of industry, government, and academic researchers with a broadly similar philosophy. The goals of HPF are to support data-parallel programming and high performance on a variety of advanced parallel architectures while minimizing deviation from the Fortran standard. HPF possesses data decomposition specifications similar to those in Fortran D, as well as additional extensions for parallel statements, intrinsic functions, extrinsic procedures, etc. We will use HPF syntax throughout this paper; however, our remarks regarding implementation are derived from our experience on the Fortran D compiler.

Generally speaking, HPF supports data-parallel programs in two ways:

- Defining operations on the elements of large data structures, such as array expressions.
- Partitioning the large data structures among the processors of a parallel machine.

The compiler is then responsible for using the data partitioning information to assign the computations to the processors.

We present only an overview of the HPF language; see (American National Standards Institute, 1990) for the full language.

3.1 Parallel Operations

HPF incorporates all of Fortran 90, including data-parallel operations such as array expressions and assignment, array intrinsics, and WHERE statements. HPF adds a number of additional data-parallel features, including new functions for reductions, combining scatters, and prefix operations. It generalizes array assignment using the `FORALL` construct and provides an `INDEPENDENT` assertion that loops may be executed in parallel. In addition to these explicit parallel features, some HPF systems automatically extract implicit parallelism from sequential constructs such as `DO` loops (Applied Parallel Research, 1992). The implementation described in Section 3.4 applies to either explicit or implicit parallelism.

HPF also supports an interface to other programming paradigms through the declaration of `EXTRINSIC` procedures. When invoked, an extrinsic procedure is executed on all processors assigned to the HPF program. Local sections of distributed arrays may be passed as parameters to the extrinsic procedure; it can perform local computations and communicate with other processors. Control returns to the HPF program when all processors exit from their extrinsic procedure.

3.2 Data Decomposition

The placement of data and computation significantly impacts performance on modern computer architectures. Therefore, to enhance efficiency, HPF augments Fortran programs with a set of data decomposition specifications. These specifications are simply hints to the compiler; if they are ignored, the program can be run without change on a sequential machine. Compilers for parallel machines can use the specifications not only to decompose data structures but also to control parallelism, based on the principle that only the owner of a datum computes its value. In other words, the data decomposition also specifies the distribution of the work in the Fortran program.

HPF approaches the data decomposition problem by noting that there are two levels of parallelism in data-parallel applications. First, there is the question of how arrays should be *aligned* with respect to one another, both within and across array dimensions. We call this the *problem mapping* induced by the structure of the underlying computation. It represents the minimal requirements for reducing data movement for the program and is largely independent of any machine considerations. The alignment of arrays in the program depends on the natural fine-grain parallelism defined by individual members of data arrays. HPF specifies this level with the `ALIGN` directive, which creates a correspondence between the elements of two arrays.

Second, there is the question of how arrays should be *distributed* onto the actual parallel machine. We call this the *machine mapping* caused by translating the problem onto the finite resources of the machine. It is dependent on the topology, communication mechanisms, size of local memory, and number of processors in the underlying machine. Data distribution provides opportunities to reduce data movement, but must also maintain load balance. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the physical parallel machine.

HPF specifies this machine mapping with the `DISTRIBUTE` directive, which divides each dimension according to a predefined pattern. The patterns available in HPF are `BLOCK` (contiguous, equal-sized ranges of elements), `CYCLIC` (every P^{th} element on the same processor, where P is the number of processors), and `*` (a dimension that is not distributed). In addition, `BLOCK` and `CYCLIC` can take a parameter giving the number of elements in a block. Figure 2 shows several possible data decompositions in HPF. We believe that this two-phase strategy for specifying data decomposition is natural for the computational scientist and is also conducive to modular, portable code.

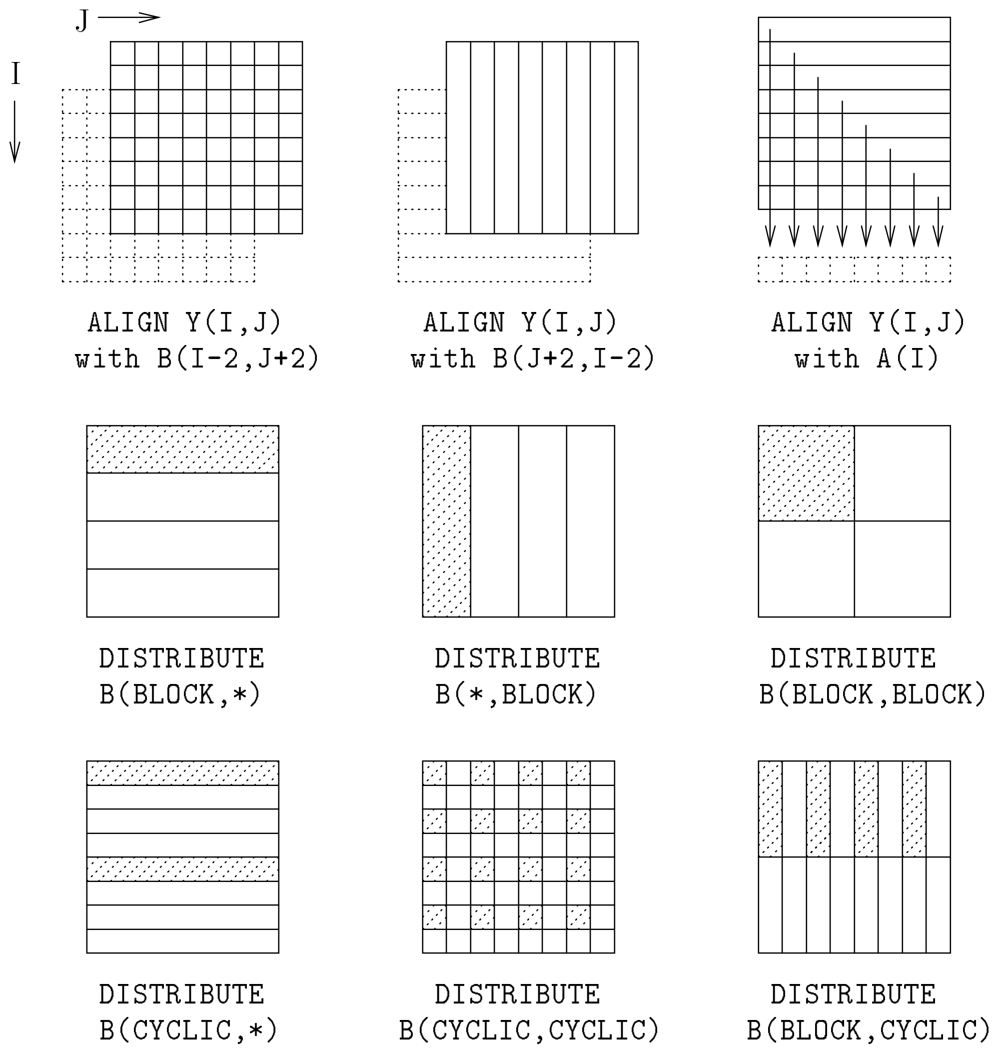

Figure 2 HPF Data Decomposition Specifications

3.3 Differences between Fortran D and HPF

Though there are many syntactic differences between Fortran D and HPF, their basic approach is quite similar. We plan to resolve these differences in the future by adapting the Fortran D compiler to accept HPF syntax. The two languages have different semantics for the `FORALL` construct, but this does not affect this paper.

Two areas explored by the Fortran D compiler are interprocedural compilation and irregular computations. To permit separate compilation, HPF provides language features that define the effect of data decomposition specifications at subprogram boundaries. In comparison, the Fortran D compiler relies on interprocedural analysis to determine data mappings across procedure boundaries. The goal is to evaluate the feasibility and usefulness of employing compiler analysis, reducing the burden on the programmer. Since Fortran D permits irregular data distributions, it also serves as a basis for investigating compiler and run-time support for irregular computations.

In summary, HPF is more carefully defined and contains features needed for supporting production-level scientific applications, while Fortran D provides a set of features designed to support interesting compiler research. We plan to continue Fortran D as a separate project to explore methods of replacing language features with advanced compiler analysis.

3.4 Implementation Issues

In order for HPF to be successful, HPF programs must be able to achieve reasonably efficient performance on a wide variety of parallel architectures. Key to meeting this requirement is the development and implementation of advanced compiler analysis and optimization. In this section we provide a brief description of HPF compilation strategy, based on experiences with the prototype Fortran D compiler at Rice University. Our philosophy is to exploit large-scale data parallelism using the *owner computes* rule, where every processor only performs computation for data it owns (Callahan and Kennedy, 1989; Rogers and Pingali, 1989; Zima, Bast, andGerndt, 1988). We also identify cases where the owner computes rule is relaxed to improve performance.

The basic approach taken by the prototype Fortran D compiler is to convert input Fortran D programs into *single-program, multiple-data* (SPMD) node programs with explicit message-passing. The two main concerns for the prototype compiler are to ensure that (i) data and computations are partitioned across processors and (ii) communication is generated where needed to access nonlocal data. The compiler applies an optimization strategy based on data dependence that incorporates and extends previous techniques. We briefly describe major steps of the compilation process below; details are presented elsewhere (Hiranandani, Kennedy, and Tseng, 1992; Hall et al., 1992; Tseng, 1993):

1. **Analyze program.** The Fortran D compiler performs scalar dataflow analysis, symbolic analysis, and dependence testing to determine the type and level of all data dependences.
2. **Partition data.** The compiler analyzes Fortran D data decomposition specifications to determine the decomposition of each array in a program. Alignment and distribution statements are used to calculate the array section owned by each processor.
3. **Partition computation.** The compiler partitions computation across processors using the "owner computes" rule—where each processor only computes values of data it owns (Callahan and Kennedy, 1988; Rogers and Pingali, 1989; Zima, Bast, and Gerndt, 1988). The left-hand side of each assignment statement is used to calculate its *local iteration set*, the set of loop iterations that cause a processor to assign to local data.
4. **Analyze communication.** Based on the computation partition, references that result in nonlocal accesses are marked.
5. **Optimize computation and communication.** Nonlocal references are examined to determine optimization opportunities. The key optimization, message vectorization, uses the level of loop-carried true dependences to extract element messages out of

```
        {* HPF Program *}
        REAL, DIMENSION(100,100) ::  A, B
!HPF$   PROCESSORS DIMENSION(4) ::  PROCS
!HPF$   ALIGN A(I,J) WITH B(I,J)
!HPF$   DISTRIBUTE A(*,BLOCK) ONTO PROCS
        A(2:99,2:99) = 0.25*(B(1:98,2:99) + B(3:100,2:99)
                           + B(2:99,1:98) + B(2:99,3:100))
                     ⇓
        {* HPF Compiler Output *}
        REAL A(100,25), B(100,0:26)
        my$p = mynode()     {* 0...3 *}
        lb$1 = max((my$p*25)+1,2)-(my$p*25)
        ub$1 = min((my$p+1)*25,99)-(my$p*25)
        if (my$p .GT. 0) send B(2:99,1) to my$p-1
        if (my$p .LT. 3) send B(2:99,25) to my$p+1
        if (my$p .GT. 0) recv B(2:99,26) from my$p+1
        if (my$p .LT. 3) recv B(2:99,0) from my$p-1
        do j = lb$1, ub$1
          do i = 2,99
            A(i,j) = 0.25*(B(i-1,j) + B(i+1,j) +
                           B(i,j-1) + B(i,j+1))
          enddo
        enddo
```

Figure 3 HPF Compilation Process

loops, combining them into less expensive vectors (Balasundaram et al., 1990; Zima, Bast, and Gerndt, 1988).

6. **Manage storage.** Buffers or "overlaps" (Zima, Bast, and Gerndt, 1988) created by extending the local array bounds are allocated to store nonlocal data.

7. **Generate code.** The compiler *instantiates* the communication, data, and computation partition determined previously, generating the SPMD program with explicit message-passing that executes directly on the nodes of the distributed-memory machine.

Figure 3 displays how a simple Jacobi solver is written as a HPF program and compiled into SPMD message-passing code for a four-processor machine. Computation is parallelized by partitioning iterations of the `i` loop across processors. Communication is extracted and combined into vector messages outside both the `i` and `j` loops. Preliminary experimental results show that the prototype Fortran D compiler can closely approach the performance of hand-optimized code for stencil computations, but requires additional improvements for linear algebra and pipelined codes (Hiranandani, Kennedy, and Tseng, 1993).

4 Integrating FM and HPF

In this section, we describe techniques that can be used to interface a task-parallel language (Fortran M) and a data-parallel language (High Performance Fortran). The interface enables the use of the task-parallel language to coordinate concurrent data-parallel computations, allows data-parallel computations to invoke task-parallel computations, and permits concurrently-executing data-parallel computations to interact by exchanging messages. The interface is conceptually simple and is expected to be useful in a wide range of scientific and engineering applications, particularly in the area of multidisciplinary design and optimization.

A key notion underlying the proposed interface is the integration of FM resource management constructs and HPF data distribution constructs. This allows a coordinating task-parallel computation to control the initial location of both distributed data and the data-parallel computations that operate on this data. For example, a coordinating task-parallel program can partition a parallel computer into several submachines and invoke a different data-parallel computation in each.

4.1 Calling HPF from FM

The ability to call HPF from FM allows FM programs to be used to coordinate concurrent execution of different data-parallel programs. Two main issues must be addressed in an FM/HPF interface: resource management and data sharing. In addition, some method of identifying data-parallel procedures to the task-parallel language compiler is needed, since different linkage conventions are needed for the different programming models. In the following, we use the statement `HPFCALL` for this purpose.

We first consider resource management. The virtual computer in which a HPF procedure is to execute may be specified via a `SUBMACHINE` annotation on the call; if none is provided, the HPF procedure executes in the same virtual computer as the calling process. Data and computation for a HPF procedure called from FM are distributed only over this virtual computer, rather than over all processors, as would be case in a pure data-parallel programming system. For example, the following code calls the HPF routines `CONTROLS` and `DYNAMICS`, invoking each in a virtual computer of size 10. Arrays `X` and `Y` are passed as arguments. (In practice, we also need to provide a mechanism by which these processes can communicate. This issue is discussed below.)

```
PROGRAM PROG1
PROCESSORS(20)
REAL X(1000,1000), Y(1000,1000)
PROCESSES
  HPFCALL CONTROLS(X) SUBMACHINE(1:10)
  HPFCALL DYNAMICS(Y) SUBMACHINE(11:20)
ENDPROCESSES
```

In contrast, the following code locates the two HPF computations on the same 10 processors:

```
PROGRAM PROG2
PROCESSORS(10)
REAL X(1000,1000), Y(1000,1000)
PROCESSES
  HPFCALL CONTROLS(X)
  HPFCALL DYNAMICS(Y)
ENDPROCESSES
```

It may happen that FM arrays passed as arguments to a HPF program are not distributed in the manner expected by the HPF program. (Indeed, they may not be distributed at all!) Hence, redistribution operations may be required to convert FM data distributions into those required for the HPF computation. Similar redistribution operations may be required upon termination of the HPF computation, prior to passing data back to the calling FM program.

In the example above, arrays `X` and `Y` are not distributed. Hence, redistribution is required to convert `X` and `Y` to whatever distribution is required in `CONTROLS` and `DYNAMICS`. This may involve considerable communication. FM programs can also define distributed arrays using HPF-style data-distribution statements (Chandy and Foster, 1993). This is illustrated in the following code. Arrays `X` and `Y` are distributed over the 10 nodes of the virtual computer in which program `PROG3` is executing. In this case, redistribution is required only if the `CONTROLS` and `DYNAMICS` computations use a different distribution.

```
PROGRAM PROG3
PROCESSORS(10)
REAL, DIMENSION(1000,1000) :: X, Y
ALIGN X(I,J) WITH Y(I,J)
DISTRIBUTE X(*,BLOCK)
PROCESSES
  HPFCALL CONTROLS(X)
  HPFCALL DYNAMICS(Y)
ENDPROCESSES
```

4.2 Calling FM from HPF

As noted previously, a data-parallel computation may include a phase that is not naturally data parallel. In a purely data-parallel programming system, this phase would execute sequentially, introducing a sequential bottleneck. Hence, we allow HPF programs to call FM routines; this permits a programmer to avoid such bottlenecks by calling FM routines that implement appropriate task-parallel algorithms.

The issues involved in the HPF/FM interface are essentially the same as those involved in the FM/HPF interface. A special notation (e.g., `FMCALL`) is required to distinguish a call to a FM procedure. The FM procedure executes on the same (virtual) processors as the calling HPF program, and redistribution operations may be required for arrays passed as arguments.

4.3 Communicating between HPF Computations

Two concurrent data-parallel computations initiated by a coordinating task-parallel computation may need to exchange data. This is possible in principle if the coordinating task-parallel computation establishes channels and passes the appropriate ports to the data-parallel program. However, it is necessary to define the mechanism by which data-parallel programs operate on these ports.

In a truly integrated task-parallel/data-parallel programming language, `SEND` and `RECEIVE` operations would be incorporated into HPF. This would require some care to avoid creating multiple writers or readers. One approach would be to designate one processor to perform all `SEND` and `RECEIVE` operations in an HPF program. While conceptually simple, this would serialize the communication to and from that process. Another possibility would be to provide parallel `INPORT` and `OUTPORT` data types and data-parallel `SEND` and `RECEIVE` statements. This would allow many processes to cooperate in sending data to other process groups, and would be particularly appropriate for communicating distributed arrays or other data structures. This approach is being explored in a compilation system under development at Argonne National Laboratory and Syracuse University (Foster et al., 1994). A disadvantage is that HPF does not currently provide such operations.

An alternative approach is to use an existing mechanism, namely, HPF's *extrinsic procedure* facility (High Performance Fortran Forum, 1993). In the HPF code, an extrinsic procedure call is made to a FM subroutine. This has the effect of calling the FM routine on every processor that is executing the HPF routine, passing the local section of the distributed array arguments on each processor, as well as copies of all scalar variables. For example, the following code calls the FM procedure `FM_PROC`. The `EXTRINSIC` declaration is needed to indicate to the compiler that `FM_PROC` is an extrinsic procedure, and thus may use a different linkage convention from that used by HPF subroutines.

```
INTERFACE
  EXTRINSIC SUBROUTINE FM_PROC(PI, Y, Z)
  INTEGER PI(10)
  REAL Y(1000)
  REAL Z
  !HPF$ DISTRIBUTE PI(BLOCK), Y(CYCLIC)
  END SUBROUTINE
END INTERFACE
...
CALL FM_PROC(A, B, C)
```

FM code called in this way can perform `SEND` and `RECEIVE` statements on ports passed as distributed array arguments. (Single ports cannot be passed as scalar arguments to HPF procedures, since these will be duplicated at every processor, defeating determinism.)

A difficulty encountered when defining this interface is that HPF does not have a port data type. In a full HPF implementation, this could be solved by defining `INPORT` and `OUTPORT` as Fortran 90 abstract data types. However, no full HPF implementation exists yet, least of all in our prototype implementations. A less satisfactory solution to this problem is to use some conventional representation of port arrays inside HPF computations, such as arrays of integers. Use of these port arrays as ordinary integers would, of course, have to be restricted by programming convention. For example, multiplying two ports together would not be meaningful.

4.4 Implementation Issues

We discuss briefly some practical issues that must be addressed when interfacing FM and HPF.

We first consider structural issues that arise when attempting to integrate existing FM and HPF compilers. FM is designed to support dynamic task-parallel computations in which the number and identify of the processors allocated to a computation may be determined only at runtime, and several computations may be mapped to the same processor. Hence, the FM runtime is multithreaded, and mapping decisions typically depend on runtime parameters. In contrast, Fortran D and HPF compilers normally assume that they are generating code for a dedicated machine of known size. The runtime code assumes a single thread per processor, and mapping decisions are made at compile time whenever possible.

These differences complicate the development of an integrated FM/HPF system based on existing compilers. A full FM/HPF programming system must allow multiple HPF and/or FM computations to execute on each physical processor as independent threads. However, existing Fortran D and HPF compilers do not support this level of generality. In a FM/HPF prototype, it may be desirable to make the restriction that only one HPF or FM node can be located on each physical processor, so as to simplify implementation.

In a full FM/HPF programming system, the size and physical location of the virtual computer within which a FM process executes are not necessarily known at compile time. Hence, a FM process must pass a representation of this virtual computer to any HPF routine that it calls, and the HPF compiler must be able to generate code that uses the processors specified by this representation. This requirement can be handled in several ways:

- Part of the linkage convention can incorporate a processor count and a vector of processor identifiers. This approach has the advantage of conceptual simplicity. However, many prototype HPF compilers (including the Rice Fortran 77D compiler) require the number of processors to be known at compile time. Such systems would require considerable rewriting to interface with task-parallel languages. Other systems, such as the Fortran 90D compiler designed at Syracuse University, rely on finely tuned library routines for efficiency. It is unclear how dynamically chosen processor sets would affect their performance.

- The FM processes calling HPF routines can be restricted in some way. For example, routines compiled for exactly 10 processors might be called unchanged from FM if the calling process were guaranteed to use 10 processors. (Some additional work might be needed to deal with the identities of the processors.) This approach has the advantage that it could be implemented quickly, but is not a viable alternative in the long run.

Similar issues arise when HPF calls FM.

A second set of issues relates to the representation of data passed between task- and data-parallel computations as arguments or in common blocks. FM and HPF compilers and runtime systems must either use common data representations or provide facilities for coverting between different representations. In addition, as discussed above, redistribution operations may be required. Interface specifications will probably be required so as to enable FM and HPF compilers to generate correct code at language boundaries without the need for interprocedural analysis.

Conventions for passing procedure arguments are also vital when HPF calls FM via the extrinsic procedure facility. In particular, every process in FM has a private memory space; in HPF, all memory is conceptually shared. This dichotomy can be addressed by providing a translation mechanism between the two paradigms. In essence, what is needed is a way to translate a FM local address (process identifier plus position in a local array) into an HPF global address (position in the global array, given the array distribution). The HPF language specification describes one possible translation mechanism (High Performance Fortran Forum, 1993), and several of the data-parallel languages in Section 5 give similar definitions.

In an aggressive optimizing compiler, however, the local to global translation may become quite complex. In particular, such compilers often define overlap areas at the edges of the array section owned by a node. Adding these to the translation (and to the memory allocation) is tedious, but necessary for correctness. Moreover, the sizes of the overlap areas may depend on the HPF code being compiled and on the compiler in use. This situation suggests that the HPF compiler must produce query routines for memory allocation in the FM calling routines. Such interactions must be built into the system from the beginning rather than as an afterthought.

A third set of issues relates to compiler analysis and optimization. HPF and HPF-like compilers rely heavily on program analysis. For example, the Rice Fortran 77D compiler performs both intra- and inter-procedural analysis to allocate memory, parallelize computation, and manage communication. This implies that the compiler needs information about all parts of the program, such as which procedure starts execution, what procedures can be called, and what data can be accessed by the called procedures. In a combined HPF/FM system, the compiler must generate this information either by analyzing both Fortran dialects directly or by relying on summary data for routines that it cannot analyze. As a first step, we propose to provide summary information regarding FM routines to the HPF compiler using auxiliary files.

Once the analysis is completed, optimization by the data-parallel compiler may also introduce complexity. We have already mentioned overlap areas as a possible interaction. Another example is interprocedural optimization of communication. Some research has shown that moving communication across procedure boundaries can have great benefits (Hall et al., 1992). However, if the compiler assumes that such optimizations are performed, the FM programmer who wants to call an HPF routine may face serious difficulties. In effect, the task-parallel programmer would have to replicate the data-parallel compiler's analysis. One approach to this problem would be to make worst-case assumptions at all FM/HPF interfaces. While limiting the compiler optimizations, this would minimize the programmer's work. Another approach would be to provide an interface explaining the HPF compiler's important assumptions and optimizations; this would at least tell the FM programmer the preconditions needed to call the HPF routine.

Another class of interactions relates to the effectiveness of the data-parallel compilers when they are combined with task-parallel languages. HPF compilers usually assume that they are generating code for a dedicated machine, or at least a dedicated partition of a shared machine. The choices of computation and communication granularity are made based on estimates of performance in the dedicated environment. Such assumptions are clearly not valid in FM's threaded environment, where multiple FM and HPF threads may execute on a single processor. No research has been done to date on the effect of such multiprogramming on code optimized by data-parallel compilers.

5 Related Work

There has been a large amount of research in the area of providing language and compiler support for either task or data parallelism. Relatively few researchers, however, have focused on the issue of integrating support for both types of parallelism. Languages for supporting or coordinating task-level parallelism include PCN (Chandy and Taylor, 1991; Foster, Olson, and Tuecke, 1992), Strand (Foster and Taylor, 1990), Linda (Carriero and Gelernter, 1989), and Delirium (Graham, Lucco, and Sharp, 1993). Tools for exploiting task-level parallelism include Schedule (Dongarra and Sorensen, 1987), Hence (Beguelin et al., 1991), and CODE (Newton and Browne, 1992). PCF Fortran exemplifies a parallel language for exploiting fork-join and loop-level parallelism (Parallel Computing Forum, 1991). In comparison with these systems, Fortran M is designed to provide greater support for modularity and safety.

Languages for expressing data-parallelism include Fortran 90 (American National Standards Institute, 1990; CMF (Thinking Machines Corporation, 1991); Vienna Fortran (Chapman, Mehrotra, and Zima, 1992), C* (Rose and Steele, 1987), Dataparallel C (Hatcher and Quinn, 1991), Dino (Rosing, Schnabel, and Weaver, 1991), Kali (Koelbel and Mehrotra, 1991), and Paragon (Chase et al., 1991). Compilers for data-parallel programs include Adapt (Merlin, 1991), Adaptor (Brandes, 1993) Aspar (Ikudome et al., 1990), Callahan and Kennedy (Callahan and Kennedy, 1988), Forge90 (Applied Parallel Research, 1992), Id Nouveau (Rogers and Pingali, 1989), and Superb (Zima, Bast, and Gerndt, 1988). The Fortran D compiler adapts techniques from several of these systems, but performs greater compile-time analysis and optimization and relies less on language extensions and run-time support.

Massingill describes mechanisms that allow the use of PCN to coordinate SPMD programs written in message-passing C (Massingill, 1993). However, this framework is more restrictive than that considered here, in that SPMD computations must execute on disjoint sets of processors and cannot communicate. Delves et al. describe an extended Fortran 90 compiler which includes both message passing along channels (as extensions to file I/O operations) and remote procedure calls (Delves et al., 1992). However, data distribution issues are not addressed in their prototype implementation.

One of the few systems to examine both types of parallelism, the iWarp compiler pursues a sophisticated compile-time approach for exploiting both task and data parallelism on a mesh-connected distributed-memory machine (Subhlok et al., 1993). The input Fortran program contains Fortran 90 array constructs, Fortran D data decomposition specifications, and parallel sections. The iWarp compiler analyzes statements and directives to produce a uniform task graph labeled with communication edges, maps each task to a processor, and inserts communications. Depending on the problem size, the best performance is obtained when exploiting task parallelism, data parallelism, or a combination of both. In comparison with the iWarp compiler, FM supports more general and dynamic forms of task parallelism, but requires the user to explicitly manage tasks and communication. The interface between FM and HPF reflects the need to support arbitrary user-specified tasks and communication.

6 Conclusions

Both task and data parallelism are important for many large scientific applications. In this paper, we have demonstrated language and compiler support for both task and data parallelism. Fortran M programs exploit task parallelism by providing language extensions for user-defined process management and typed communication channels. Fortran D and High Performance Fortran (HPF) programs exploit data parallelism by providing language extensions for user-defined data decomposition specifications, parallel loops, and parallel array operations. Preliminary experiences show that both programming models are portable yet efficient. We have also presented the design of an interface for integrating support for both task and data parallelism. We believe future experiences will demonstrate its usefulness.

References

American National Standards Institutte 1990. ANSI X3J3/S8.115. Fortran 90.

Applied Parallel Research 1992. *Forge 90 distributed memory parallelizer: User's guide*, version 8.0 edition, Placerville, California: Applied Parallel Research.

Balasundaram, V., Fox, J., Kennedy, K., and Kremer, U. 1990. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th distributed memory computing conference*, Charleston, S.C.

Beguelin, A., Dongarra, J., Geist, G., Manchek, R., and Sunderam, V. 1991. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing '91*, Albuquerque, N.M.

Brandes, T. 1993. Automatic translation of data parallel programs to message passing programs. In *Proceedings of AP'93 international workshop on automatic distributed memory parallelization, automatic data distribution and automatic parallel performance prediction*, Saarbrücken, Germany.

Callahan, D., and Kennedy, K. 1988. Compiling programs for distributed-memory multiprocessors. *J. Supercomputing* 2:151–169.

Carriero, N., and Gelernter, D. 1989. Linda in context. *Comm. ACM* 32(4):444–458.

Chandy, K. M., and Foster, I. 1993. Deterministic parallel Fortran. In *Proceedings of the sixth SIAM conference on parallel processing for scientific computing*, Norfolk, Virginia.

Chandy, K. M., and Taylor, S. 1991. *An introduction to parallel programming.* Jones and Bartlett.

Chapman, B., Mehrotra, P., and Zima, H. 1992. Programming in Vienna Fortran. *Scientific Programming* 1(1):31–50.

Chase, C., Cheung, A., Reeves, A., and Smith, M. 1991. Paragon: A parallel programming environment for scientific applications using communication structures. In *Proceedings of the 1991 international conference on parallel processing*, St. Charles, Illinois.

Delves, L., Craven, P., and Lloyd, D. 1992. A Fortran 90 compilation system for distributed memory architectures. In *Proceedings PACA92 conference*, IOP.

Dongarra, J., and Sorensen, D. 1987. SCHEDULE: Tools for developing and analyzing parallel Fortran programs. In *The characteristics of parallel algorithms*, edited by D. Gannon, L. Jamieson, and R. Douglass. Cambridge, Massachusetts: The MIT Press.

Foster, I., Avalani, B., Choudhary, A., and Xu., M. A compilation system that integrates High Performance Fortran and Fortran M. In *Proc. 1994 Scalable High Performance Computing Conf.*, Knoxville, Tenn., 1994.

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

Foster, I., Olson, R., and Tuecke, S. 1992. Productive parallel programming: The PCN approach. *Scientific Programming* 1(1):51–66.

Foster, I., Olson, R., and Tuecke, S. 1993. Programming in Fortran M. Technical Report ANL-93/26, Argonne National Laboratory.

Foster, I., and Taylor, S. 1990. *Strand: New concepts in parallel programming.* Englewood Cliffs, N.J.: Prentice-Hall.

Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C., and Wu, M. 1990. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University.

Graham, S., Lucco, S., and Sharp, O. 1993. Orchestrating interactions among parallel computations. In *Proceedings of the SIGPLAN '93 conference on program language design and implementation*, Albuquerque, N.M.

Federal HPCC Program 1993. Workshop and Conference on Grand Challenges Applications and Software Technology. GCW-0593. Pittsburgh, Pa.

Hall, M. W., Hiranandani, S., Kennedy, K., and Tseng, C. 1992. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, Minn.

Hatcher, P., and Quinn, M. 1991. *Data-parallel programming on MIMD computers.* Cambridge, Mass.: The MIT Press.

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

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

Hiranandani, S., Kennedy, K., and Tseng, C. 1992. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM international conference on supercomputing*, Washington, D.C.

Hiranandani, S., Kennedy, K., and Tseng, C. 1993. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, Oregon.

Ikudome, K., Fox, G., Kolawa, A., and Flower, J. 1990. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the 5th distributed memory computing conference*, Charleston, S.C.

Knobe, K., Lukas, J., and Steele, Jr., G. 1990. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *J. Parallel and Distributed Computing* 8(2):102–118.

Koelbel, C., and Mehrotra, P. 1991. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. Parallel and Distributed Systems* 2(4):440–451.

Massingill, B. 1993. Integrating task and data parallelism. TR CS-TR-93-01. Pasadena: California Institute of Technology.

Merlin, J. 1991. ADAPTing Fortran-90 array programs for distributed memory architectures. In *First international conference of the Austrian Center for Parallel Computation*, Salzburg, Austria.

Newton, P., and Browne, J. C. 1992. The CODE 2.0 graphical parallel programming language. In *Proceedings of the 1992 ACM international conference on supercomputing*, Washington, D.C.

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

Parallel Computing Forum 1991. PCF: Parallel Fortran extensions. *Fortran Forum* 10(3).

Rogers, A., and Pingali, K. 1989. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 conference on program language design and implementation*, Portland, Oregon.

Rose, J., and Steele, Jr., G. 1987. C*: An extended C language for data parallel programming. In *Proceedings of the second international conference on supercomputing*, edited by L. Kartashev and S. Kartashev. Santa Clara.

Rosing, M., Schnabel, R., and Weaver, R. 1991. The DINO parallel programming language. *J. Parallel and Distributed Computing* 13(1):30–42.

Subhlok, J., Stichnoth, J., O'Hallaron, D., and Gross, T. 1993. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the fourth ACM SIGPLAN symposium on principles and practice of parallel programming*, San Diego.

Thinking Machines Corporation 1991. *CM Fortran reference manual*, version 1.0 edition. Cambridge, Mass.: Thinking Machines Corp.

Tseng, C. 1993. *An optimizing Fortran D compiler for MIMD distributed-memory machines.* Ph.D. thesis. Dept. of Computer Science, Rice University.

Zima, H., Bast, H.-J., and Gerndt, M. 1988. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing* 6:1–18.