PREPRINT MCS-P422-0394, MATHEMATICS AND COMPUTER SCIENCE DIVISION,
ARGONNE NATIONAL LABORATORY

LIBRARIES FOR PARALLEL PARADIGM INTEGRATION

Ian Foster and Ming Xu

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439, U.S.A.

ABSTRACT

A programming paradigm is a method for structuring programs in order to reduce the complexity of the programming task. In parallel programming, task and data parallelism are the most common paradigms, although object, functional, logic, and database parallelism are also used. Most existing programming languages and tools are designed to support either task parallelism or data parallelism. Yet there are many applications that can benefit from a combination of both paradigms. For example, image processing problems can exploit the regularity of data-parallel computation within each stage of a pipeline. This process requires tools that support both task and data parallelism and that allow task- and data-parallel components to interact. In particular, it should be possible to reuse libraries developed in either paradigm in a variety of situations. To meet these requirements, we have designed a small set of extensions to Fortran, called Fortran M, that provides a framework on which portable, reusable libraries can be built to implement programming paradigms. In this paper, we illustrate the approach by showing how it is used to implement libraries for file I/O, message-passing, and data-parallel computation.

1. INTRODUCTION

Most existing parallel programming tools are designed to support a single-program multiple-data (SPMD) or *data-parallel* programming model, in which each processor executes the same program but operates on different data. This model is fundamental to data-parallel languages such as C* and High Performance Fortran and accounts for the bulk of programs written in message passing libraries such as MPI, p4, and

PVM. The advantages of the data-parallel approach include regular communication patterns and ease of programming and debugging.

Nevertheless, an increasing number of parallel applications have a richer structure in which threads of control may execute different programs and may be created and deleted dynamically. This is the case, for example, in multidisciplinary simulation, in which a complex system (such as an aircraft) is constructed by coupling models of system components (such as fluid dynamics, structural mechanics, surface heating, and controls). Similarly, image-processing algorithms often have a heterogeneous pipeline structure that is not naturally represented in an SPMD context. These applications require what is often called *task* parallelism.

Data and task parallelism are often seen as opposing approaches to programming. Yet in practice we find that applications frequently include both task- and data-parallel components. For example, although the interactions between the components of a multidisciplinary simulation involves task parallelism, individual components may well admit to a data-parallel solution. In these and other applications, the ability to integrate multiple paradigms in a single program has the potential for improving both programmer productivity and performance. A lack of tools, however, makes the development of such *multiparadigm* applications difficult in practice.

In this paper, we introduce an approach to multiparadigm programming based on language extensions. The idea is to define a small set of extensions to an existing sequential language that can be used to specify concurrency, communication, synchronization, and mapping. They also provide what we call *compositionality*, meaning that program components encapsulating concurrency, communication, and mapping decisions can be reused in different situations without concern for internal implementation details. The extended sequential language is then used to write *paradigm libraries* that support various paradigms. Thanks to compositionality, paradigm libraries can be integrated in flexible ways to implement multiparadigm programs. In this paper, we use an extended Fortran called Fortran M (FM) [6] to illustrate the approach; however, the basic ideas are language independent. In a related effort, colleagues at Caltech are using an extended C++ for the same purpose [3].

To explore the utility of the FM extensions for multiparadigm programming, we have been working with colleagues to develop libraries for the integration of message passing, for data parallel programming [5, 2], for redirection of file I/O, for scientific programming "templates," and for exploitation of parallelism in automatic differentiation [1]. In this paper, we describe the mechanisms used to support the first three of these applications.

2. FORTRAN M

Fortran M (FM) is a language designed by researchers at Argonne and Caltech for expressing task-parallel computation [6]. 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.

FM is currently defined as extensions to Fortran 77; however, equivalent extensions can easily be defined for Fortran 90. For clarity, we use some Fortran 90 syntax in

```
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 structures(qi,po)
endprocesses
end


process controls(inp,outp)
inport (integer, real x(10,20), real y(10,20)) inp
outport (integer, integer, real x(10,10,3)) outp
...
send(outp) i, j, a
receive(inp) nstep, u, v
...
end
```

Figure 1: Sketch of a Multidisciplinary Program, Using FM

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. 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 `structures`. These execute concurrently, on the same or different 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 a FM compiler to generate efficient communication code and to convert to a machine-independent format in a heterogeneous environment.

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 the `submachine` annotation specifies that a process should execute in a subset of the array. For example, the following code places the `controls` and `structures` processes on different virtual processors.

```
processors(2)
...
processes
  processcall controls(...) location(1)
  processcall structures(...) 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 using a SPMD library.

```
processors(20)
...
processes
  processcall controls(...) submachine(1:10)
  processcall structures(...) submachine(11:20)
endprocesses
```

3. FILE INPUT/OUPUT LIBRARIES

Our first example of a paradigm library is almost trivial but nevertheless useful. This is a file I/O compatibility library (FIOCL). With FIOCL, programs that were designed originally to interact with their environment by reading and writing files can be integrated into a task-parallel framework by reinterpreting file read and write operations as receive and send operations on channels. The library uses FM processes to encapsulate different programs and channels to represent the "virtual files" on which these programs perform read and write operations.

A program that is to be integrated into an FM framework must be modified to incorporate calls to FIOCL subroutines `assoc_inport` and `assoc_outport`. These associate a Fortran file with an FM inport or outport, respectively, and cause subsequent read or write operations on the file to be implemented as receive and send operations. The program must also be converted into a process. It can then be plugged together with other programs by FM code that creates the required channels and passes the associated inports and outports to these programs.

We illustrate the approach with an example similar to that used in the preceding section. Figure 2 sketches versions of programs `controls` and `structure` that have been adapted to operate by using the FM file I/O library. The two programs are assumed to read and write file data, respectively. To allow the two programs to operate concurrently, we add to both programs a `process` declaration and add three statements (indicated by comments) that declare a port, include some definitions, and specify that the file operated on by the program is to be associated with a port. Subsequent open, read, and write operations then operate on the specified files. The code that sets up the necessary channel and invokes the two components is also shown.

We have not yet discussed the mechanism by which I/O statements are made to operate on ports rather than files. In a language such as C, I/O operations are implemented as library calls; hence, it would suffice to link application programs with a modified I/O library that checked each I/O operation to see whether it concerned a file or a channel. As this approach is not possible in Fortran, we introduce a simple preprocessor that replaces I/O calls with calls to library routines. For example, a `write` statement is replaced with "`call fiocl_write`".

4. MESSAGE-PASSING LIBRARIES

We now describe a message-passing compatibility library (MPCL) that allows programs developed using message-passing (MP) libraries (e.g., p4, PVM, Express) to be integrated into a more general task-parallel framework. This allows MP programs to be invoked from FM and permits MP programs to call FM routines. The invoking program can use FM virtual computer constructs to specify the resources (virtual processors) that will be available to the MP program; the MP program executes as if these virtual processors were physical processors, and performs ordinary MP calls. Various 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. Figure 3 illustrates some of these ideas. The enclosing oval represents a FM process, with local data structures `X` and `Y`. The enclosed execution graph represents initiation of an MP program on four virtual processors, with argument `X` scattered and `Y` replicated.

MPCL consists of two components: an interface routine that sets up the N^2 channels required to allow N processes to communicate with each other, and then invokes

```
      program aerodynamics
      inport (integer n, character buff(n)) pi
      outport (integer n, character buff(n)) po
      channel(in=pi,out=po)              ! Create channel
      processes                          ! Create processes
        processcall controls(pi)
        processcall structures(po)
      endprocesses
      end

      process controls(pi)
      inport (...) pi                    ! Inport for communication
      include 'fio.inc'                  ! File I/O library definitions
      call associnport('indata',pi)      ! Associate file with inport
      ...
      open(unit=11, file='indata')
      ...
      read(11,99) a, b, c
99    format(3F10.2)
      ...
      end

      process structures(po)
      outport (...) po                   ! Outport for communication
      include 'fio.inc'                  ! File I/O library definitions
      call assocoutport('outdata', po)   ! Associate file with outport
      ...
      open(unit=9, file='outdata')
      ...
      write(9,99) a, b, c
99    format(3F10.2)
      ...
      end
```

Figure 2: Using the File I/O Library

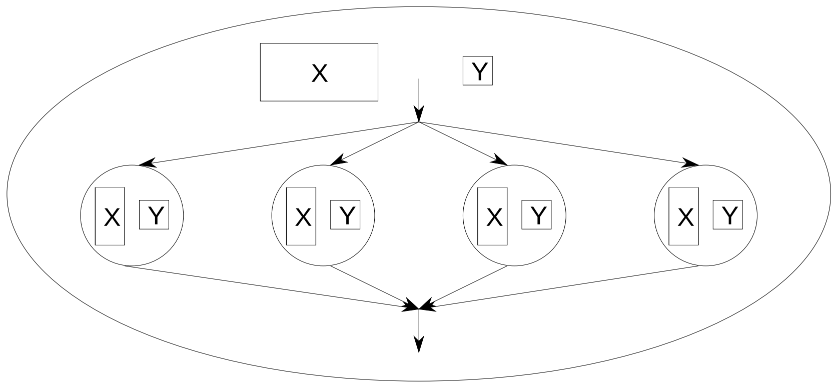

Figure 3: Invoking a Message-Passing Program from FM

an MP program in N processes; and a set of FM implementations of a particular MP library's routines (e.g., `p4send`, `p4recv`, etc., in p4). MPCL currently supports subsets of the p4, PICL, and Express libraries.

The interface code has a structure similar to that shown in Figure 4. This example program is intended to initiate execution of the MP program `prog`, passing FM variables `X` and `Y` as arguments (partitioned and replicated, respectively). It creates `N` instances of a process `prog_wrap`, passing each an array of `N` outports, which can be used to send messages to the other processes, and a single inport, on which messages from other processes are received. The `N` outports destined for a single process are associated with that process's inport by the nondeterministic `MERGER` construct. The subroutine `prog_wrap` stores the ports and other information passed as arguments in a common block (defined in the include file `mp.inc`) and then invokes `prog`.

The FM implementations of the MP library routines use the channel structure created by the initialization routine to implement send, receive, and other operations. For example, the following is an implementation of a generic `mp_send` operation that allows the caller to specify a tag, message size, message contents, and destination. The array `outps` contains the outports passed as arguments to the `prog_wrap` call and stored in a common block by `stash_ports`.

```
subroutine mp_send(tag,size,msg,dest)
integer tag, size, dest              ! Message tag, size, dest.
character msg(size)                  ! The data to be communicated
include 'mp.inc'                     ! Include file with channels
send (outps(dest)) tag, size, msg    ! Send message to dest
return
end
```

Although the interface code in Figure 4 is not especially complicated, it would nevertheless be useful if we could encapsulate it in a channel setup and message-passing setup library. This library would have to be parameterized with the name of

```
real x(N,M), y
outport (integer n, character buff(n)) outps(N,N)
inport (integer n, character buff(n)) incps(N)
do i=1,N
   merger(in=inps(i),out=(outps(i,j),j=1,N))
enddo
processdo i = 1,N
   processcall prog_wrap(i,N,outps(i,1:N), inp(i), x(i,1:M), y)
endprocessdo

process prog_wrap(num,outps,inp,arg1, arg2)
outport (integer n, character buff(n)) outps(N,N)
inport (integer n, character buff(n)) inps(N)
real arg1(M), arg2
include 'mp.inc'
call stash_ports(num,inp,outps)
call prog(arg1,arg2)
enddo
```

Figure 4: FM Wrapper Code for Message-Passing Program

the message-passing process (in our case, `prog`) and the number and distribution of the user arguments passed (e.g., `X` and `Y`). Unfortunately, since Fortran 77 supports neither higher-order functions nor variable-length argument lists, a parameterized library of this sort is not possible.

We circumvent this problem by providing a preprocessor that generates the required interface code. This preprocessor allows the user to use an `mpcall` statement to invoke an MP program and a `scatter` statement to specify that an array variable should be partitioned when passed as an argument to an MP program. (`BLOCK` distributions in one or two dimensions are currently supported.) Together, these features allow us to replace Figure 4 with the following statements.

```
real x(N,M), y
scatter x(BLOCK,*)
mpcall prog(x,y)
```

The use of these features is further illustrated in the following code fragment, which invokes two MP programs, `controls` and `structures`, one after the other, passing the variable `X` as an argument to both.

```
program main
processors pr(64)
real x(512, 512)
scatter x(*,BLOCK)
...
mpcall controls(x)
```

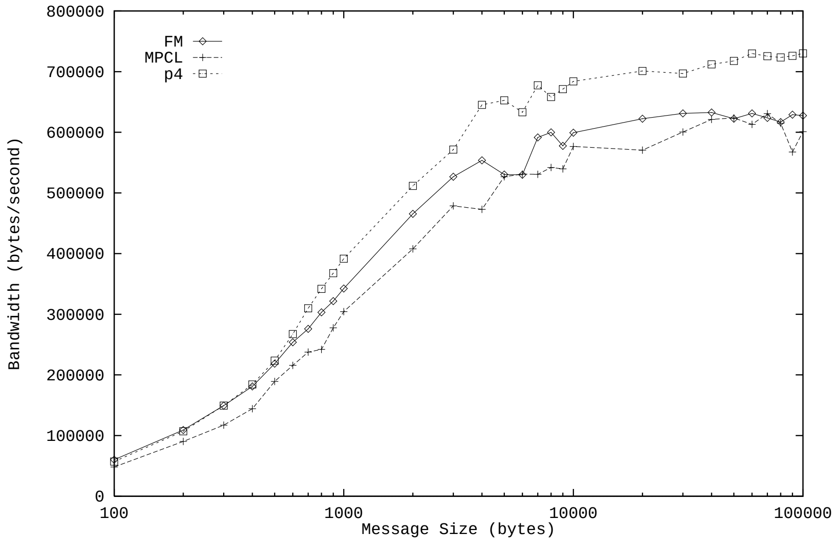

Figure 5: Bandwidth of FM, MPCL, and p4 on Ethernet

```
...
mpcall structures(x)
```

We are conducting detailed performance studies to determine whether this library-based approach to the integration of message-passing programs leads to significant overhead. Some preliminary results are presented in Figure 5. This shows achieved bandwidth, as measured by timing a simple program that bounces messages of varying size between two processors, on two Ethernet-connected Sparc-2 workstations. Results are given for versions of the program written in FM, in p4 using the compatibility library (MPCL), and in raw p4. We see that FM and p4 provide comparable performance for smaller messages, but that p4 is slightly faster for larger messages. MPCL is consistently about ten per cent slower than FM or p4. We expect that this overhead (which is quite acceptable for many applications) can be reduced significantly.

5. HIGH PERFORMANCE FORTRAN

The mechanisms described in the preceding section can also be used to integrate programs written in High Performance Fortran (HPF). HPF compilers typically compile HPF programs to Fortran plus calls to message-passing libraries. Hence, a simple integration strategy is first to compile the HPF program using an HPF compiler and then to link the resulting code with the FM message-passing library. We have pursued this approach successfully in collaboration with Bhaven Avalani and Alok Choudhary of Syracuse University [5]. This allows us to write code similar to the following. This code fragment implements an image-processing convolution pipeline, in which two input image streams flow through two forward fast Fourier transform (FFT) stages, and then through a multiplication and inverse FFT stage. We have demonstrated this application on the IBM SP1 and shown that this mixed task/data-parallel formulation can provide performance superior to that of a purely data-parallel program.

```
C       FM Main Program for Mixed Algorithm
        program mixed_parallel
        processors pr(24)
        outport (complex x(512,512/8)) outs1(8), outs2(8)
        inport (complex x(512,512/8)) ins1(8), ins2(8)
        channel(in=ins1(:), out=outs1(:))
        channel(in=ins2(:), out=outs2(:))
        processes
          mpcall fft(nimages, outs1) submachine(1:8)
          mpcall fft(nimages, outs2) submachine(9:16)
          mpcall ifft(ins1, ins2) submachine(17:24)
        endprocesses
```

This approach allows the integration of HPF procedures into FM task-parallel programs. FM data structures can be passed as arguments to HPF procedures. This requires either that the FM library has knowledge of the data structures used in the HPF compiler or that the HPF compiler is aware of the format of the FM data structures. We have adopted the former approach.

FM port variables can also be passed as arguments to HPF procedures, providing in principle a mechanism by which HPF programs can communicate with other, concurrently executing, HPF or FM procedures. This can be achieved without extensions to the HPF compiler by (a) representing these variables within HPF in terms of the representation used by the FM compiler (which happens to be integers), and (b) using the HPF extrinsic procedure mechanism to invoke FM procedures that operate on these variables. This approach is simple but inelegant. An alternative approach is to extend HPF with port data types and data-parallel send and receive operations that send and receive arrays and scalars on arrays of outports and inports, respectively. We are currently pursuing this approach in collaboration with Syracuse.

6. RELATED WORK

Two alternative approaches to multiparadigm programming are new languages and libraries. New languages can provide direct support for both the SPMD and task-parallel programming models; however, they can require considerable investment compiler development, code conversion, and user training. In the library approach, paradigm libraries make direct calls to low-level primitives for task creation, communication, and synchronization, rather than using language extensions. This avoids the need for compiler development but lacks desirable properties such as compositionality and compile-time checking.

More limited in scope are approaches based on the use of a task-parallel coordination language able to invoke data-parallel computations. 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 the approach described here. 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 [10]. 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. The use of FM and data-parallel paradigm libraries permits more general and dynamic forms of task parallelism but also requires more programmer intervention.

7. CONCLUSIONS

We have used three simple examples to show how the FM extensions to Fortran can be used to develop libraries that support the integration of programs developed using diverse paradigms into a single task-parallel framework. These libraries use FM processes to encapsulate message-passing computation, channels to implement communication, and virtual computer constructs to control resource allocation. The resulting programs can be reused in any environment.

In general, we prefer to support the integration of a new paradigm by means of a library rather than additional language extensions. The limitations of Fortran syntax, however, often encourage us to use simple source transformations to facilitate the use of a particular paradigm. In some cases, these transformations are essential (FIOCL); in others, they are cosmetic (MPCL, HPF). These extensions are incorporated in a preprocessor; if they prove to be genuinely useful, we might eventually incorporate them into our FM compiler.

Our future plans include the development of Fortran 90 language extensions to support multiparadigm programs; the more modern constructs provided by Fortran 90, including structured data types and dynamic memory allocation, can support a more powerful set of primitives. We are also investigating compiler and runtime system optimization techniques and the integration of paradigm libraries written in different languages. In the latter area, a particular focus is integration with the CC++ language under development at Caltech [3]. This task is simplified by the fact that CC++ and FM use the same runtime system.

More information on this work is available via Mosaic at the following address:

`http://www.mcs.anl.gov/fortran-m/FM.html`.

In addition, a FM compiler is available from anonymous ftp server `info.mcs.anl.gov`, in directory `pub/fortran-m`.

ACKNOWLEDGMENTS

This research was supported by the NSF Center for Research in Parallel Computation (CRPC) Contract CCR-8809615 and by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38. We are grateful to other members of the CRPC's Paradigm Integration group, particularly Mani Chandy, Carl Kesselman, Robert Olson, and Steven Tuecke, for their many contributions to this work. Dennis Gannon's SAGE project at Indiana University provided transformation tools used in this work.

References

[1] C. Bischof, L. Green, K.Haigler, and T. Knauff, Parallel Calculation of Sensitivity Derivatives for Aircraft Design Using Automatic Differentiation, Preprint MCS-P419-0294, Argonne National Laboratory, Argonne, Ill., 1994.

[2] Chandy, K. M., 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] Chandy, K. M., 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

[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] Foster, I., 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 (to appear), 1994.

[6] Foster, I., and K. M. Chandy, Fortran M: A Language for Modular Parallel Programming, Preprint MCS-P327-0992, Argonne National Laboratory, Argonne, Ill., 1992.

[7] Fox, G., S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, Fortran D Language Specification, Technical Report TR90-141, Computer Science, Rice Univ., Houston, Texas, 1990.

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

[9] 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, 1993.

[10] 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.