FORTRAN M: A Language for Modular Parallel Programming

Ian T. Foster
K. Mani Chandy

CRPC-TR92249
October 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

computers, networks of workstations, and other parallel computer architectures. Its popularity stems from its simplicity, flexibility, and ease of implementation.

A disadvantage of the message-passing model, particularly for scientific and engineering applications, is that it does not enforce deterministic execution [22]. Hence, the programmer has no *a priori* guarantee that a program will give the same result if executed more than once with the same input. This nondeterminism is antithetical to both the scientist's need for reproducibility and ease of debugging. In addition, most message passing systems do not enforce information hiding and provide a global name space of processes. This makes it difficult to develop modular programs and reusable libraries [10].

In this paper, we describe message-passing extensions to sequential programming languages that enforce both deterministic execution and information hiding, while retaining much of the flexibility of traditional message-passing. We describe these extensions in the context of FORTRAN 77, and call the resulting language FORTRAN M. However, equivalent extensions can be defined for any sequential programming language. The extensions include 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. The resulting programming model is dynamic: processes and channels can be created and deleted dynamically, and references to channels can be included in messages.

FORTRAN M enforces determinancy by means of syntactic and semantic restrictions. In addition, a FORTRAN M compiler can use type information provided by the programmer to verify correct usage. The price of this safety is that the programmer must explicitly declare and create the communication channels that will be used in a program. However, this requirement appears no more onerous than variable type declarations, which serve a similar purpose. FORTRAN M also provides nondeterministic constructs for programs that operate in nondeterministic environments. The use of these constructs is typically isolated to a small number of modules.

FORTRAN M enforces information hiding, and hence facilitates a modular or *object-oriented* approach to parallel program design. In particular, it permits the definition of reusable program components. A channel is only accessible to a process that possesses a reference to it. Common data is only supported on a per-process basis. Hence, a process's interface to its environment is defined by the channels passed to it as arguments. All other details of its implementation, which can include common data, subprocesses, process placement, and internal communication channels, are hidden.

FORTRAN M is supported by a theory of parallel and sequential composition of communicating processes. Key characteristics of this theory, described in a separate paper [3], include (1) proofs that a FORTRAN M program is deterministic even though processes and channels are created and deleted and channels are reconnected; (2) extension of sequential programming proof techniques to parallel programs; and (3) a compositional proof theory in which the specification of the whole is derived from the specifications (and not the texts) of the part.

The basic paradigm underlying FORTRAN M is *task parallelism*: the parallel execution of (possibly dissimilar) tasks. Hence, FORTRAN M complements *data-parallel* languages such as FORTRAN D [14] and High Performance FORTRAN (HPF). In particular, FORTRAN M can be used to coordinate multiple data-parallel computations. Our goal is to integrate HPF with FORTRAN M, thus combining the data-parallel and task-parallel programming paradigms in a single system. This integration is facilitated by support for data distribution statements as in HPF, which allow distributed arrays to be declared within FORTRAN M and then operated on in parallel by either HPF or FORTRAN M procedures.

In the rest of this paper, we define FORTRAN M and illustrate its application to programming problems. In Sections 2 and 3, we present the constructs used to define and compose processes. In Sections 4–8, we discuss dynamic process and communication structures, nondeterministic constructs, argument passing, process placement, and data distribution. Sections 9 and 10 discuss compilation and related work. We conclude in Section 11.

A prototype FORTRAN M compiler for sequential and parallel computers is available by anonymous ftp from Argonne National Laboratory, in directory pub/pcn on machine `info.mcs.anl.gov` [12].

2 Defining Modules

In modular program design, we develop components of a program separately, as independent modules, and then combine modules to obtain a complete program [24, 7]. Interactions between modules are restricted to well-defined interfaces. Hence, module implementations can be changed without modifying other components, and the properties of a program can be determined from the specifications for its modules and the code that plugs these modules together. When successfully applied, modular design reduces program complexity and facilitates code reuse.

In FORTRAN M, a module is implemented as a *process*. A process, like a FORTRAN program, defines common data (labeled `PROCESS COMMON` to emphasize that it is local to the process) and the subroutines that operate on that data. It also defines the interface by which it communicates with its environment. A process has the same syntax as a subroutine, except that the keyword `PROCESS` is used in place of `SUBROUTINE`.

2.1 Interfaces

A process's dummy arguments (formal parameters) are a set of *port variables*. These define the process's interface to its environment. (For convenience, conventional argument passing is also permitted between a process and its parent. This nonessential feature is discussed in Section 6.) A port variable declaration has the general form

port_type (*data_type_list*) *name_list*

The *port_type* is `OUTPORT` or `INPORT` and specifies whether the port is to be used to send or receive data, respectively. The *data_type_list* is a comma-separated list of type declarations. It specifies the format of the messages that will be sent on the port, much as a subroutine's dummy argument declarations defines the arguments that will be passed to the subroutine.

For example, the following process declares in-ports capable of receiving messages consisting of single integers (p1), arrays of `MSGSIZE` reals (p2), and a single integer and a real array with size specified by the integer (p3). In the third declaration, the names m and x have scope local to the port declaration.

```
process example(p1,p2,p3)
parameter(MSGSIZE=20)
inport (integer) p1
inport (real x(MSGSIZE)) p2
inport (integer m, real x(m)) p3
```

We illustrate the use of ports with a simple example. A program that simulates the atmospheric circulation (an atmosphere model) is to be coupled with an ocean model. The two models are to execute concurrently and must exchange information periodically: The ocean model provides the atmosphere model with an array of sea surface temperatures (SST), and the atmosphere model provides the ocean model with two arrays containing components of horizontal momentum, U and V. We implement both models as processes, and define an interface that allows for the exchange of SST, U, and V values.

We assume initially that the atmosphere model is a sequential program. (A parallel version is presented in the next section.) Hence, we define an interface consisting of two ports, **sstin** and **uvout**. The in-port **sstin** can be used to receive arrays of real values representing sea surface temperatures, while the out-port **uvout** can be used to send two such arrays representing U and V values.

```
process atmosphere(sstin,uvout)
parameter(NLAT=128,NLON=256)
inport  (real x(NLAT,NLON)) sstin
outport (real x(NLAT,NLON), real y(NLAT,NLON)) uvout
...
```

2.2 Communication

As each process has its own address space, the only mechanism by which a process can interact with its environment is via the ports passed to it as arguments. A process uses the SEND, ENDCHANNEL, and RECEIVE statements to send and receive messages on these ports. These statements are similar in syntax and semantics to FORTRAN's WRITE, ENDFILE, and READ statements, and can include END=, ERR=, and IOSTAT= specifiers to indicate how to recover from various exceptional conditions.

A process sends a message by applying the SEND statement to an out-port. The out-port declaration specifies the message format. A process sends a sequence of messages by repeated calls to SEND; it can also call ENDCHANNEL to send an end-of-channel (EOC) message. The SEND and ENDCHANNEL statements are nonblocking (asynchronous): they complete immediately. A process receives a value by applying the RECEIVE statement to an in-port. A RECEIVE statement is blocking (synchronous): it does not complete until data is available.

For example, the following code repeatedly sends U and V data on the port **uvout** and receives SST data from the port **sstin**. After doing this TMAX times, it signals the end of the communication by sending an EOC message on **uvout**. For illustrative purposes, we use process common to hold the **sst**, u, and v arrays.

```
      process atmosphere(sstin,uvout)
      parameter(NLAT=128, NLON=256, TMAX=100)
C     The ports are the external interface.
      inport (real x(NLAT,NLON)) sstin
      outport (real x(NLAT,NLON), real y(NLAT,NLON)) uvout
C     Process common variables.
      process common /state/ sst, u, v
      real sst(NLAT,NLON), u(NLAT,NLON), v(NLAT,NLON)
      call init
C     Repeat TMAX times: send U & V, recv SST, update U & V.
      do 10 i=1,TMAX
        send(uvout) u,v
        receive(sstin) sst
        call atm_compute
10    continue
C     Signal end of communication.
      endchannel(uvout)
      end
```

The ocean model can be defined in a similar fashion, with an out-port `sstout` for SST data and an in-port `uvin` for U and V data. This process repeatedly sends SST data on `sstout` and receives U and V data on `uvin`, until EOC is detected on `uvin`.

3 Composing Modules

A FORTRAN M program is constructed by using *process blocks* and *process do-loops* to plug together (compose) processes. A program creates *channels* to establish one-to-one communication streams between processes. In this way, processes with more complex behaviors are defined. These can themselves be composed with other processes, in a hierarchical fashion.

3.1 Composition of Processes

A process block is a form of parbegin/parend [8], with the general form

```
processes
  statement_1
  ...
  statement_n
endprocesses
```

where $n \geq 0$, and the statements are process calls (distinguished by the keyword PROCESSCALL), process do-loops (defined below), and/or at most one subroutine call. Statements in a process block execute *concurrently*. For example, the following block specifies that the processes `atmosphere` and `ocean` are to execute concurrently.

```
processes
  processcall atmosphere(...)
  processcall ocean(...)
endprocesses
```

A process block terminates, allowing execution to proceed to the next executable statement, when all of its constituent statements terminate.

3.2 Channels

Recall that a process communicates with its environment by sending and receiving messages on ports. When composing processes, we use the CHANNEL statement to define these ports to be references to first-in/first-out message queues called *channels*. This statement has the general form

CHANNEL(out=*out-port*, in=*in-port*)

and both creates a channel and defines *out-port* and *in-port* to be references to this channel. These ports are to be used for sending and receiving messages, respectively, and can be passed as arguments to the composed processes.

In the ocean/atmosphere model, we require two channels, one for communicating SST values and the other for communicating U and V values. This structure is illustrated in Figure 1 and is created by the following program. Note that this code defines a process; if ports are added to define an interface, it can be combined with other processes to form a more complex program. The process creates two channels, spawns the atmosphere and ocean processes, blocks until the process block terminates, and then terminates itself.

```
      process coupled_model
      parameter(NLAT=128, NLON=256)
C     Local port variables.
      inport (real x(NLAT,NLON)) sstin
      outport (real x(NLAT,NLON)) sstout
      inport (real x(NLAT,NLON), real y(NLAT,NLON)) uvin
      outport (real x(NLAT,NLON), real y(NLAT,NLON)) uvout
C     Create channels and define ports.
      channel(out=ssto, in=sstin)
      channel(out=uvout, in=uvin)
C     Call two models with ports as arguments.
      processes
        processcall atmosphere(sstin,uvout)
        processcall ocean(uvin,sstout)
      endprocesses
      end
```

The value of the four port variables declared in this code fragment is initially undefined. The CHANNEL statements each create a channel and define their two port variable

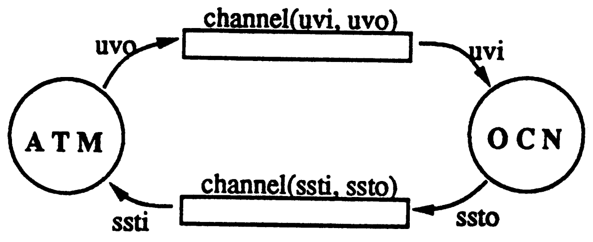

Figure 1: Coupled Ocean/Atmosphere Model

arguments to be references to this channel. These port variables are passed as arguments to the concurrently executing `atmosphere` and `ocean` processes, establishing the connections shown in Figure 1.

We now have a complete parallel program which can be executed on a sequential or parallel computer. We shall see that this program can be executed on one processor or two without any change to its component modules. The execution order of the concurrently executing `atmosphere` and `ocean` processes is determined only by availability of messages on channels. Nevertheless, the computed result does not depend on the order in which the processes execute. That is, the program is deterministic.

3.3 Replicating Processes

A process do-loop creates multiple instances of the same process. It is frequently used to define single program, multiple data (SPMD) computation structures, in which multiple copies of a process are connected in a regular communication structure. The process do-loop is identical in form to the do-loop, except that the keyword `PROCESSDO` is used in place of `DO` and the body can include only a process do-loop or a process call. For example:

```
   processdo 10 i = 1,n
     processcall myprocess
10 continue
```

Process do-loops can be nested inside both process do-loops and process blocks.

We illustrate the use of the process do-loop in Program 1, which implements a parallel version of the atmosphere model. The parallel code partitions the model's data domain into `NP` subdomains of size (`NLAT`) $\times$ (`PLON=NLON/NP`) and uses 2 `NP` channels to connect these processes in a ring. Figure 2 shows the original grid, the decomposition, and the process structure, with `NLAT=6`, `NLON=12` and `NP=2`.

Two arrays of ports, `NIn` and `NOut`, are declared and then defined to be references to the 2 `NP` channels. Each `subdomain` process is passed four of these ports; these provide in and out connections to its two neighbors.

It is desirable to provide a parallel interface to a parallel model, so that components of the model can communicate with corresponding components of other parallel models

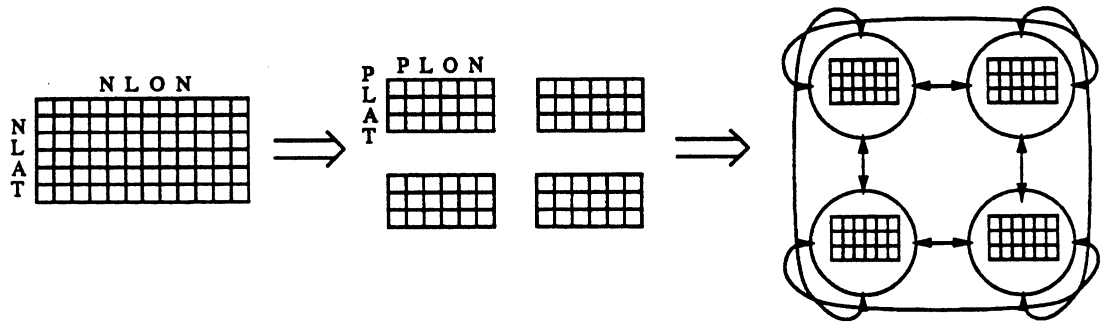

Figure 2: Parallel Atmosphere Model

without introducing a central bottleneck. Hence, the interface to the parallel model is also decomposed, giving two arrays of ports, `SstIn` and `UvOut`, each of size `NP`. Each port in these arrays is used to communicate arrays of size `NLAT`×`PLON`. Each `subdomain` process is passed one element of `SstIn` and one element of `UvOut` as arguments.

The code used to compose the atmosphere and ocean models must be modified as follows to allow for the parallel interface. The two channels `sstout/sstin` and `uvout/uvin` are replaced with arrays of `NP` channels, and the calls to the sequential processes are replaced with calls to the parallel processes.

```
      program coupled_model
      parameter(NLAT=128,NLON=256,NP=16,PLON=NLON/NP)
      inport (real x(NLAT,PLON)) SstIn(NP)
      outport (real x(NLAT,PLON)) SstOut(NP)
      inport (real x(NLAT,PLON), real y(NLAT,PLON)) UvIn(NP)
      outport (real x(NLAT,PLON), real y(NLAT,PLON)) UvOut(NP)
      ...
C     Create NP channels.
      do 10 i=1,NP
        channel(out=SstOut(i), in=SstIn(i))
        channel(out=UvOut(i), in=UvIn(i))
10    continue
      ...
C     Pass port arrays to parallel models.
      processes
        processcall par_atmosphere(SstIn,UvOut)
        processcall par_ocean(SstOut,UvIn)
      endprocesses
      end
      process par_atmosphere(SstIn,UvOut)
      parameter(NLAT=128,NLON=256,NP=16,PLON=NLON/NP)
C     These two port arrays define external interface.
      inport (real x(NLAT,PLON)) SstIn(NP)
      outport (real x(NLAT,PLON), real y(NLAT,PLON)) UvOut(NP)
C     Ports for communication with W and E neighbors.
      inport (real x(NLAT)) FromNbr(2,NP)
      outport (real x(NLAT)) ToNbr(2,NP)
C     Create channels used for internal communication.
      do 10 i = 1,NP
        channel(in=FromNbr(2,i), out=ToNbr(1,mod(i,NP)+1))
        channel(out=ToNbr(2,i), in=FromNbr(1,mod(i,NP)+1))
10    continue
C     Create NP processes, with external and internal ports.
      processdo 20 i = 1,NP
        processcall subdomain(SstIn(i), UvOut(i), ToNbr(1,i),
                              ToNbr(2,i), FromNbr(1,i), FromNbr(2,i))
20    continue
      end


C     Code executed in a single subdomain.
      process subdomain(sstin,uvout,ToW,ToE,FromW,FromE)
      parameter(NLAT=128,NLON=256,NP=16,PLON=NLON/NP)
C     External interface ports: for sending SST and receiving U & V.
      inport (real x(NLAT,PLON)) sstin
      outport (real x(NLAT,PLON), real y(NLAT,PLON)) uvout
C     Ports to and from W and E neighbors.
      inport (real x(NLAT)) FromW, FromE
      outport (real x(NLAT)) ToW, ToE
      ...
```

Program 1: Parallel Atmosphere Model

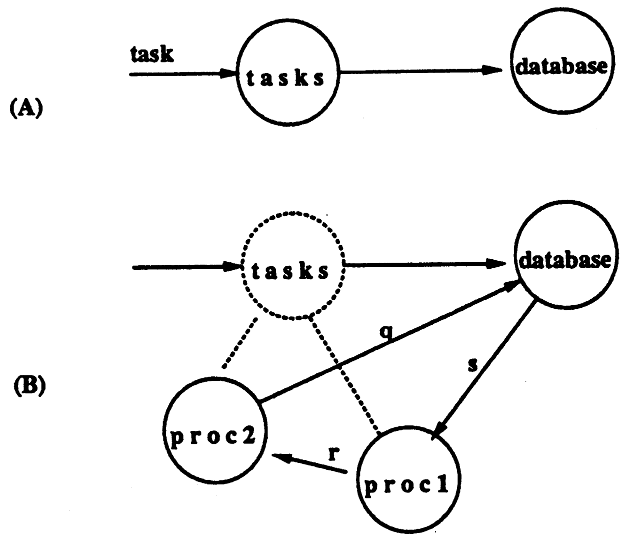

Figure 3: A Dynamic Process and Communication Structure

4 Dynamic Structures

The process and communication structures in the ocean/atmosphere model are essentially static: after an initial startup phase, the number of processes and channels does not change. FORTRAN M can also be used to specify dynamic structures in which processes and channels are created and deleted, and channels are reconnected, during the course of a computation.

This is illustrated in the following example. Consider a process network consisting of a tasks and a database process, as illustrated in Figure 3(A). The tasks process receives a sequence of integers representing tasks on its in-port. Each time it receives an integer, it creates three new channels and sends ports referencing two of these channels to database. It then establishes the process structure illustrated in Figure 3(B), by creating a proc1 and a proc2 process and passing the appropriate ports to these processes as arguments. The proc1, proc2, and database processes communicate among themselves until proc1 and proc2 terminate. Then, the network reverts to that shown in Figure 3(A), and tasks handles the next incoming message.

This structure is specified as follows. Note the declaration of the out-port po, which specifies that the port is used to transmit messages consisting of an integer, an integer out-port, and an integer in-port. Each time a task is received, three channels are created and qi/qo, ri/ro, and si/so are defined to be references to these channels. Two of these ports, qi and so, are sent to the database process; the remaining ports are passed as arguments to proc1 and proc2.

```
      process tasks(mi,po)
C     Ports defining external interface.
      inport (integer) mi
      outport (integer, outport (integer), inport(integer)) po
```

```
C    Ports for local communication.
     inport (integer) qi, ri, si
     outport (integer) qo, ro, so
C    Repeat: receive a task, create 3 channels, send ports on po.
     do while(.true.)
       receive(mi) task
       channel(out=qo, in=qi)
       channel(out=ro, in=ri)
       channel(out=so, in=si)
       send(po) task,qi,so
       processes
         processcall proc1(si,ro)
         processcall proc2(ri,qo)
       endprocesses
     enddo
     end
```

The ability to transfer a channel reference from one process to another is useful but potentially dangerous. If not controlled, it could compromise determinism by permitting multiple out-ports to reference the same channel. Hence, FORTRAN M semantics ensure that only a single copy of a channel reference can exist at any one time. When the contents of a port variable are communicated in a message, the value of that port variable becomes undefined. Similarly, assignment of one port variable to another is not permitted; the MOVEPORT statement must be used to copy a port, and this makes the copied port variable undefined. Hence, execution of the following code fragment, which stores the value of mi in ri and sends the value of qi on the port po, causes both mi and qi to become undefined.

```
inport  (integer) qi, mi, ri
outport (inport (integer) ) po
moveport(from=mi, to=ri)
send(po) qi
```

5 Nondeterminism

The determinism enforced by the use of channels removes a major source of complexity in concurrent programming. However, nondeterminism can be useful in nondeterministic environments. For example, a load-balancing algorithm may need to execute either a local or remote task, depending on which is the first to become available. Similarly, we may wish to process requests to access a shared data structure, or input from several external devices, in the order in which they become available. These behaviors can be specified by using the MERGE and PROBE statements.

A MERGE statement defines a first-in/first-out message queue, just like a CHANNEL statement. However, it allows multiple out-ports to reference this queue and hence defines a many-to-one communication structure. Messages sent on any out-port are appended to the queue, with the order of messages sent on each out-port being preserved and any message sent on an out-port eventually appearing in the queue.

For example, consider the following problem, proposed to us by Burton Smith. NP monte_carlo processes execute independently and generate integer "scores" at irregular intervals. We wish to generate a histogram of these values. One possible solution is to create a single histo process and use MERGE to link the out-ports of the monte_carlo processes and the in-port of the histo process. This solution can be implemented as follows.

```
      program histogram
      parameter(NP=128)
      inport (integer) pi
      outport (integer) Po(NP)
C     The merger links all out-ports with the in-port.
      merge(out=(Po(i),i=1,NP),in=pi)
      processes
        processcall histo(pi)
        processdo 10 i = 1,NP
          processcall monte_carlo(Po(i))
10      continue
      endprocesses
      end
```

An alternative, less centralized solution to the problem can also be specified in a straightforward manner. The histogram is distributed among many histo processes, and mergers are used to connect each histo process with all monte_carlo processes. A quadratic number of ports must be declared if messages are to be routed in constant time. If NP is large, the program can be modified to utilize a communication network of lower dimension, at the cost of additional communication steps.

A process can apply the PROBE statement to an in-port to determine whether messages are pending on a channel. It sets a logical variable, specified in an EMPTY=*variable* specifier, to true if the channel is empty and to false otherwise.

6 Argument Passing

In preceding programming examples, all communication between processes has occurred via ports. For programming convenience, FORTRAN M also allows conventional argument passing between a process and the processes that it calls (its children). The values of these arguments are passed to a child processes when they are created, and copied back to the parent process when the children terminate. Copying is performed in textual and iteration-count order in order to ensure deterministic execution. A child process can also specify, by INTENT declarations, that particular arguments are to be copied on call or return only. For example, the following process has three input arguments and one output argument. It computes an approximation to the integral of a function F(x) over the range a.h $\leq x \leq$ b.h using the rectangle rule and interval h. That is, it computes $\sum_{j=a+1}^{b} F((j-0.5)*h)$.

```
        process integrate(idx_a,idx_b,h,sum)
        intent(in)  idx_a, idx_b, h
        intent(out) sum
        sum = 0.0
        do 10 i=idx_a+1,idx_b
          sum = sum + F((i-0.5)*h)
   10   continue
        end
```

A dummy argument declared INTENT(IN) is copied only when the process is called. If no intent declaration is provided for a dummy argument, or it is declared INTENT(INOUT), then the corresponding actual argument, which must be a variable, is updated after the process terminates. For a dummy argument declared INTENT(OUT), the corresponding actual argument must also be a variable, and its value is again updated upon process termination. However, in this case the variable is set to some arbitrary value upon entry to the process.

This process is used in the following program, which computes an approximation to the integral of $F(X)$ over the interval (0,1). (For comparison, solutions to the same problem in several other parallel FORTRAN dialects are presented in [17].) The process creates NP `integrate` processes, each of which evaluates the integral over a specified subinterval and stores its result in an element of the array `results`. Upon termination of the `processdo` statement, elements of this array are summed by the main program.

```
        program integration
        parameter(NP=128)
        real results(NP)
        read(*,*) intvls
        icomps = intvls/NP
        if(icomps*NP .ne. intvls) stop(99)
        processdo 10 i=1,NP
          processcall integrate((i-1)*icomps,i*icomps,1.0/intvls,results(i))
   10   continue
        sum = 0.0
        do 20 i = 1,NP
          sum = sum + results(i)
   20   continue
        print *,'Sum is ',sum/intvls
        end
```

7 Process Placement

Process blocks and do-loops define concurrent processes; channels and mergers define how these processes communicate and synchronize. A parallel program defined in terms of these constructs can be executed on both uniprocessor and multiprocessor computers. In the latter case, processes must be mapped to processors.

The techniques used to map processes to processors depends in part on the architecture of the parallel computer in question. If a small number of processors share access to a common memory, then automatic mechanisms — based, for example, on a centralized scheduler — may be effective. However, the importance of the memory hierarchy in larger parallel computers means that process placement (mapping) can be an important aspect of algorithm design. For this reason, FORTRAN M incorporates constructs that allow mapping to be specified by the programmer. 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.

7.1 Process Placement Constructs

The FORTRAN M 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 topology as the physical computer on which a program executes [20, 30]. For consistency with FORTRAN concepts, a FORTRAN M virtual computer is an N-dimensional array, and the mapping constructs are modeled on FORTRAN 77's 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 [10].

The `PROCESSORS` declaration is similar in form and function to the array `DIMENSION` statement. It has the general form `PROCESSORS`$(\mathtt{I}_1,\ldots,\mathtt{I}_n)$ where $n \geq 0$ and the $\mathtt{I}_j$ have the same form as the arguments to a `DIMENSION` statement. It specifies the shape and size of the (implicit) processor array on which a process is executing. This processor array cannot be larger than that declared in the parent, but it can be smaller or of a different shape.

The `LOCATION` annotation is similar in form and function to an array reference. It has the general form `LOCATION`$(\mathtt{I}_1, \ldots, \mathtt{I}_n)$, where $n \geq 0$ and the $\mathtt{I}_j$ have the same form as the indices in an array reference, and specifies the processor on which the annotated process is to execute. The indices must not reference a processor array element that is outside the bounds specified by the `PROCESSORS` declaration provided in the process or subroutine in which the annotation occurs.

A `SUBMACHINE` annotation is similar in form and function to an array reference passed as an argument to a subroutine. It has the general form `SUBMACHINE`$(\mathtt{I}_1,\ldots,\mathtt{I}_n)$, where $n \geq 0$ and the $\mathtt{I}_j$ have the same form as the indices in an array reference. It specifies that the annotated process is to execute in a virtual computer comprising the processors taken from the current virtual computer, starting with the specified processor and proceeding in array element order. The size and shape of the new virtual computer is as specified by the `PROCESSORS` declaration in the process definition.

7.2 Mapping Examples

We specify mapping in Program 1 by providing a `PROCESSORS` declaration at the top of the program and a `LOCATION` annotation on the call to `subdomain`:

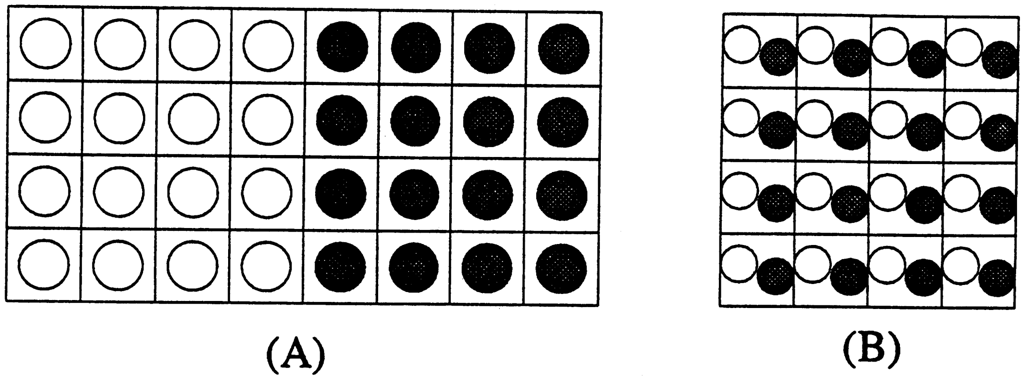
Figure 4: Alternative Mapping Strategies

```
      processors(NP)
      ...
      processdo 10 i = 1,NP
        processcall subdomain(SstIn(i), UvOut(i), NOut(1,i),
                              NOut(2,i), NIn(1,i), NIn(2,i))  location(i)
 10   continue
```

The SUBMACHINE annotation can be used to create several disjoint virtual computers, each comprising a subset of available processors. For example, in the ocean/atmosphere model, it may be desirable to execute the two models in parallel, on different parts of the same computer. This organization is illustrated in Figure 4(A) and can be specified as follows. The atmosphere model is executed in one half of a computer, and the ocean model in the other half.

```
        parameter(NP=4)
        processors(2*NP)
        ...
        processes
          processcall par_atmosphere(SstIn,UvOut) submachine(1)
          processcall par_ocean(SstOut,UvIn)      submachine(NP+1)
        endprocesses
```

Alternatively, it may be more efficient to map both models to the same set of processors, as illustrated in Figure 4(B). This can be achieved by changing the PROCESSORS declaration to PROCESSORS(NP) and omitting the SUBMACHINE annotations. No change to the component programs is required.

8 Data Parallelism

The basic paradigm underlying FORTRAN M is *task parallelism*: the parallel execution of (possibly dissimilar) tasks. However, FORTRAN M also provides some support for data-parallel computation. Programs can use data distribution statements to create distributed arrays. 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 cause communication.

We utilize High Performance FORTRAN-style data distribution statements to create distributed arrays in FORTRAN M. These statements allow FORTRAN M to specify certain classes of data-parallel computations. For example, in the following code fragment, the same computation is performed on each column of a distributed array. The `PROCESSORS` statement indicates that the program is to be compiled for an array of `N` (virtual) processors; the `LOCATION` annotation on the call to `computesum` specifies that the process is to execute on the `i`th processor.

```
processors(N)
real A(N,N), sum(N)
distribute A(*,CYCLIC), sum(CYCLIC)
processdo i=1,N
  processcall computesum(N, A(1,i), sum(i)) location(i)
enddo
end

process computesum(N, B, sum)
real B(N), sum
intent(in) N, B
intent(out) sum
sum = 0.0
do i = 1,N
  sum = sum + B(i)
enddo
end
```

9 Compilation

A prototype FORTRAN M compiler has been developed for sequential, parallel, and networked computers [12]. This compiler supports all language constructs except those concerned with data distribution, and has been used to develop parallel programs in coupled climate modeling, multidisciplinary design, computational biology, and air quality modeling, among other areas.

The FORTRAN subset of FORTRAN M is compiled with conventional compilers and thus achieves the same performance as pure FORTRAN. Hence, the primary difficulty that arises when constructing a FORTRAN M compiler is achieving efficient implementations of communication and process management mechanisms. We first consider communication. FORTRAN M's `SEND` and `RECEIVE` operations can be translated into memory-

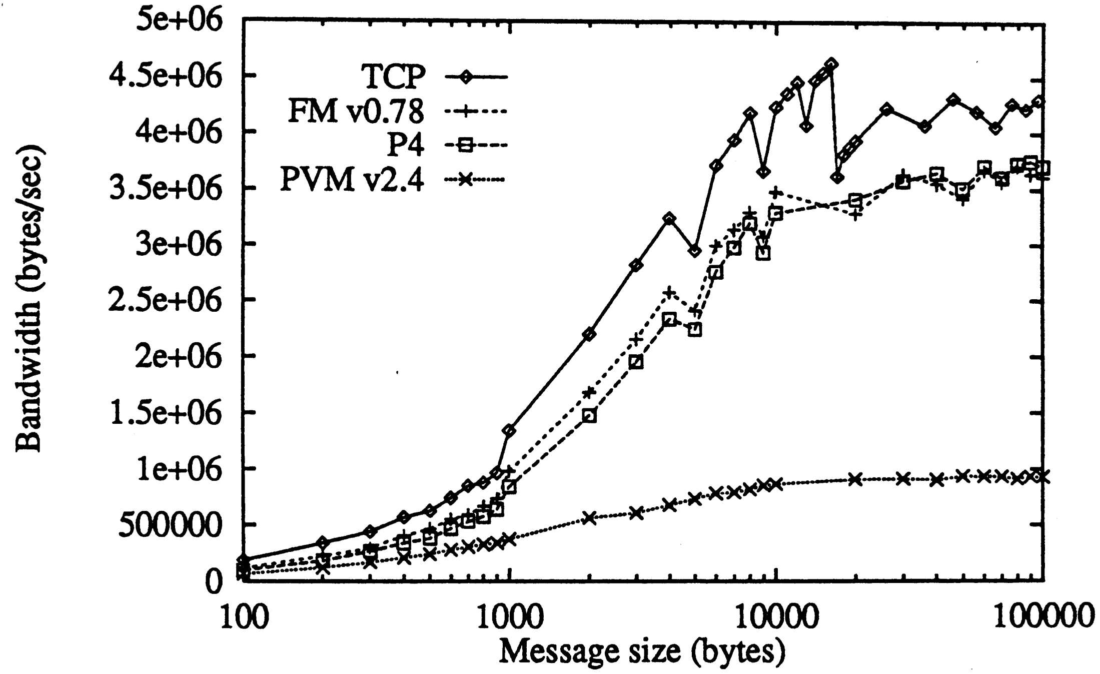

Figure 5: Interprocessor Communication Bandwidth on IBM SP1

to-memory transfers in shared-memory computers and uniprocessors and into low-level message-passing operations on distributed-memory computers. As the origin and destination of the data to be communicated are stated explicitly in send and receive calls, messages can be assembled without an intermediate packing step, if this is permitted by the underlying communication system. Message handling code at the receiver can also be simplified. Hence, we would expect the communication performance of a FORTRAN M compiler to at least equal that of a good message-passing library. We evaluate communication performance using a simple test program that creates two processes connected by channels, which then exchange a large number of fixed-size messages. Execution time is measured for different message sizes and achieved bandwidth is computed. Figure 5 presents performance results for a network environment in which processes exchange messages by using the TCP/IP sockets protocol. The results were obtained on the IBM SP1, a multicomputer based on the RIOS 1 microprocessor. Results are given for equivalent programs written in FORTRAN M; raw TCP/IP protocol; P4 [1], a highly-optimized message-passing library; and PVM v2.4 [29]. The P4 and PVM programs were provided by the P4 developers and by PVM users, respectively. We see that FORTRAN M is 20% faster than P4 for small messages and significantly faster than PVM for all message sizes. The low-level protocol is faster than FORTRAN M, but only by a small factor; this is primarily because it does not incorporate logic to decode messages on receipt. These results suggest that FORTRAN M semantics allow generation of efficient communication code.

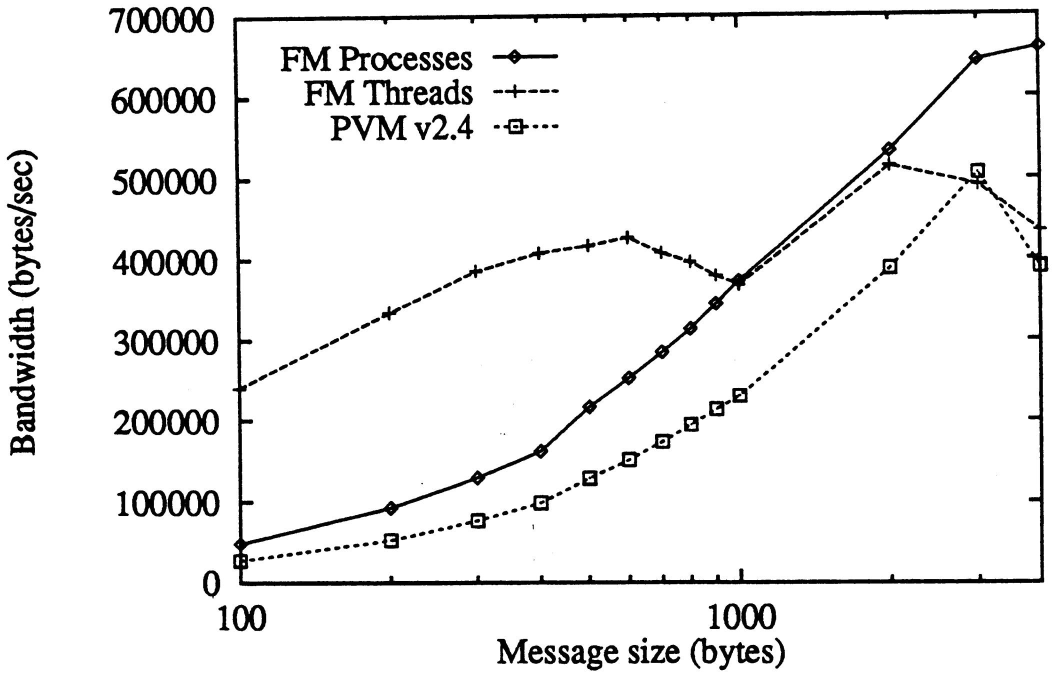

Figure 6: Intraprocessor Communication Bandwidth on Sun-4 Workstation

The performance of programs that generate multiple processes per processor depends also on the cost of process scheduling operations. Our second performance study uses the same test program as the first, but executes it on a single Sun-4 workstation. Figure 6 shows achieved *intra*processor communication bandwidth as a function of message size using two versions of the FORTRAN M compiler that target (a) Sun lightweight processes (threads) and shared memory, and (a) Unix processes and sockets. Intraprocessor results for PVM v2.4 are also shown for comparison. The thread numbers are significantly higher for smaller messages, partly because of lower process switching overheads (sending an empty message takes 0.37 msec in the thread version but 1.8 msec in the process version) and partly because shared memory rather than sockets are used for communication. Clearly, there are substantial benefits to the former, lower-level mechanisms. In the future, we plan to explore active message libraries [32], specialized thread libraries, and compile-time scheduling techniques. These developments complicate compilation: for example, we must generate reentrant code in a thread-based implementation, and specialized message handlers in an active message implementation. However, these are issues which, while difficult to handle in a message-passing library, can reasonably be addressed in a compiler.

In summary, preliminary work on FORTRAN M compilation shows that it is possible to generate code that is competitive in performance with conventional message-passing libraries. We have also demonstrated that it is possible to enhance performance in specific situations (e.g., intraprocessor communication) by exploiting lower-level mechanisms. The goal of future work will be to exploit compiler analyses and transformations to improve both communication and process management performance.

10 Related Work

Programming notations for parallel scientific programming fall into three principal classes: coordination languages, message-passing libraries, and data parallel extensions. Here, we discuss how FORTRAN M differs from each of these approaches, focusing in particular on the issues of modularity and safety. We do not consider systems based on shared-memory models [17], as these are not easily adapted to distributed-memory machines.

In coordination language approaches, a specialized language is used to specify concurrency, communication, and synchronization; FORTRAN routines are called to perform computation. This approach has the advantage of clearly separating parallel and sequential computation, but requires the programmer to learn a new language. Coordination languages include occam [16], Strand [13], PCN [5, 11], and Delerium [19]. Delerium is purely a coordination language, while the others can be used to specify both coordination and computation. occam, derived from Hoare's Communicating Sequential Processes (CSP) [15], can specify only static computation and communication structures, does not enforce determinism, and employs synchronous communication. Strand and PCN can specify dynamic structures. Communication and synchronization are specified in terms of read and write operations on single-assignment variables, and a form of guarded command is used to specify choice between alternatives. A compiler cannot in general assert that a Strand or PCN program is deterministic, because it cannot always prove that choices in guarded commands are mutually exclusive. In contrast, a compiler need only verify that a program uses neither `MERGE` nor `PROBE`. Strand and PCN do not address the problem of FORTRAN common data.

In message-passing library approaches, programmers call subroutines to communicate data between processes. The number of processes is often fixed at one per physical processor. Systems such as P4 [1], Express [23], PVM [29], and Zipcode [28] provide, as primitives, an asynchronous send to a named process and a synchronous receive. The Mach operating system provides, in addition, a virtual channel construct (the port); ports can be transferred between processes in messages [33]. Mach does not restrict copying of ports, so determinism is not enforced. Libraries have the advantage of simplicity: they are language independent and do not require compiler modifications. This simplicity comes at a price, however. Compile-time checking for correct usage is not performed. As library writers know nothing about how routines will be used, they must program defensively and incorporate logic that can, in principle, be avoided in code generated by a FORTRAN M compiler. In contrast to FORTRAN M, message-passing libraries are nondeterministic and, as the name space of processes is global, do not enforce information hiding.

Related to both coordination languages and message-passing libraries is Linda, which provides read and write operations on a shared *tuple space* [2]. Tuple space operations can emulate both message-passing communication protocols and shared data structures. Tuple space operations, like message passing, are nondeterministic and do not enforce information hiding. Actor-based message passing systems such as CE/RK [27] have some points of similarity with FORTRAN M, but are fundamentally different in that they are nondeterministic. CC++ is a shared virtual memory extension of C++ [4]. It differs from FORTRAN M in many respects, in particular its use of a shared-memory programming model.

In data parallel approaches, sequential languages are extended with directives that specify how arrays are to be decomposed and distributed over processors [31, 14, 6]. A compiler then partitions the computation using the "owner computes" rule, with each operation in the sequential program allocated to the processor containing the data that is to be operated on. This approach permits succinct specifications of parallel algorithms for regular problems and guarantees deterministic execution. When extended with support for irregular data distributions, data parallel languages can also handle some irregular problems [18, 26]. However, there are broad classes of problems for which the approach has not yet been shown to be tractable. These include highly dynamic adaptive grid problems, multidisciplinary optimization problems, and reactive systems in which a program interacts with an external environment in a nondeterministic manner. These problems can all be implemented in a straightforward manner with FORTRAN M.

11 Conclusions

High-level languages such as FORTRAN and C have been adopted almost universally in sequential programming, and for good reasons: compared with machine languages, they permit more concise specifications, more compile-time checking, and greater portability and modularity. In addition, modern compilers generate better object code than do most programmers.

For a variety of reasons, parallel computers are still programmed primarily in parallel "machine languages": locks and semaphores on shared-memory computers, and primitive send and receive operations on distributed-memory computers. Our goal in defining FORTRAN M is to make the advantages of high-level languages available to programmers developing programs for parallel machines. In particular, we are concerned with ensuring *safety*. This is achieved in two ways. First, we define language extensions that allow deterministic execution to be guaranteed. This means that programmers can be confident that their programs will produce the same output for all executions with a given input. Second, we require that the user provide type information, which a compiler can use to detect erroneous programs at compile time.

FORTRAN M's extensions to FORTRAN 77 (summarized in Figure 7) can be described in a few minutes and mastered in a few hours. The extensions have a FORTRAN 77 "look and feel". For instance, the `CHANNEL`, `SEND`, `RECEIVE`, and `ENDCHANNEL` statements are similar to `OPEN`, `WRITE`, `READ`, and `ENDFILE`. Likewise, the process placement statements are modeled on FORTRAN 77 array manipulation constructs. The extensions allow programmers to develop parallel programs by plugging together modules that encapsulate both code and data. This object-oriented approach to program design supports the implementation of reusable parallel libraries and multidisciplinary applications. Furthermore, because the extensions can be implemented efficiently on a wide variety of parallel com-

Process:	`PROCESS`
	`PROCESS COMMON`
	`PROCESSCALL`
Interface:	`INPORT`
	`OUTPORT`
Control:	`PROCESSES/ENDPROCESSES`
	`PROCESSDO`
Communication:	`CHANNEL`
	`MERGER`
	`SEND`
	`RECEIVE`
	`ENDCHANNEL`
	`MOVEPORT`
	`PROBE`
Performance:	`PROCESSORS`
	`LOCATION`
	`SUBMACHINE`
	`TEMPLATE`
	`ALIGN`
	`DISTRIBUTE`

Figure 7: FORTRAN M's Extensions to FORTRAN 77

puters, application portability is achieved with little or no performance penalty. We have demonstrated performance parity with message-passing libraries in a prototype compiler, and are currently investigating compiler analyses and transformations with the goal of realizing further performance improvements.

The definition of FORTRAN M opens several avenues for future research. The integration of data-parallel notations such as High Performance FORTRAN (HPF) with FORTRAN M *allows the implementation of heterogeneous applications, in which a FORTRAN M* program coordinates multiple data-parallel computations. Data-parallel subroutines can be invoked in a specified processor array, with ports used for communication with FORTRAN M computations. Also of interest are more general models of process interaction, that nevertheless preserve determinism and modularity [12].

Acknowledgments

We are grateful to Robert Olson and Steven Tuecke for their splendid efforts developing the prototype FORTRAN M compiler, to Andrew Lavery and John Thayer for preparing test and benchmark programs, and to Ming Xu for performance results. Ken Kennedy, John Mellor-Crummey, and others at Rice University provided valuable comments on earlier versions of this material. This research was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the National Science Foundation's Center for Research in Parallel Computation, under Contract CCR-8809615.

References

[1] Boyle, J., Butler, R., Disz, T., Glickfeld, B., Lusk, E., Overbeek, R., Patterson, J., and Stevens, R., *Portable Programs for Parallel Processors*, Holt, Rinehart, and Winston, 1987.

[2] Carriero, N., and Gelernter, D., *How to Write Parallel Programs*, MIT Press, 1990.

[3] Chandy, K. M., and Foster, I., A deterministic notation for cooperating processes, Preprint MCS-P346-0193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1993.

[4] Chandy, K. M., and Kesselman, C., Compositional parallel programming in CC++, Technical Report, Caltech, 1992.

[5] Chandy, K. M. and Taylor, S., *An Introduction to Parallel Programming*, Jones and Bartlett, 1991.

[6] Chapman, B., Mehrotra, P., and Zima, H., Vienna FORTRAN — A FORTRAN language extension for distributed memory systems, *Languages, Compilers, and Run-time Environments for Distributed Memory Machines*, Elsevier Press, 1992.

[7] Cox, B., and Novobilski, A., *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1991.

[8] Dijkstra, E.W., Cooperating sequential processes, *Programming Languages*, Academic Press, 1968.

[9] Dongarra, J., van de Geijn, R., and Walker, D., A look at scalable dense linear algebra libraries, *Proc. 1992 Scalable High Performance Computers Conf.*, IEEE Press, 992.

[10] Foster, I., Information hiding in parallel programs, Preprint MCS-P290-0292, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

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

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

[13] Foster, I. and Taylor, S., *Strand: New Concepts in Parallel Programming*, Prentice-Hall, 1989.

[14] Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C., and Wu, M., FORTRAN D language specification, Technical Report TR90-141, Department of Computer Science, Rice University, Houston, Texas, 1990.

[15] Hoare, C., Communicating Sequential Processes, *CACM*, 21(8), 666–677, 1978.

[16] Inmos, Ltd, *occam Programming Manual*, Prentice Hall, 1984.

[17] Karp, A., and Babb, R., A comparison of 12 parallel FORTRAN dialects, *IEEE Software*, 5(5), 52–67, 1988.

[18] Koelbel, C., Mehrotra, P., and Van Rosendale, J., Supporting shared data structures on distributed memory machines, *Proc. 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ACM, 1990.

[19] Lucco, S., and Sharp, O., Parallel programming with coordination structures, *Proc. 18th ACM POPL*, ACM, 1991.

[20] Martin, A., The torus: An exercise in constructing a processing surface, *Proc. Conf. on VLSI*, Caltech, 52–57, 1979.

[21] Metcalf, M., and Reid, J., FORTRAN *90 Explained*, Oxford Science Publications, 1990.

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

[23] Parasoft Corporation, Express user manual, 1989.

[24] Parnas, D., On the criteria to be used in decomposing systems into modules, *CACM,* 15(12), 1053-1058, 1972.

[25] *Programming Language* FORTRAN, American National Standard X3.9-1978, American National Standards Institute, 1978.

[26] Saltz, J., Berryman, H., and Wu, J., Multiprocessors and run-time compilation, ICASE Report 90-59, Institute for Computer Applications in Science and Engineering, Hampton, Virginia, 1990.

[27] Seitz, C. L., Multicomputers, *Developments in Concurrency and Communication*, Addison-Wesley, 1991.

[28] Skjellum, A., and Leung, A., Zipcode: A portable multicomputer communication library atop the Reactive Kernel, *Proc. 5th Distributed Memory Computer Conf.*, IEEE Press, 767-776, 1990.

[29] Sunderam, V., PVM: A framework for parallel distributed computing, *Concurrency: Practice and Experience*, 2, 315–339, 1990.

[30] Taylor, S., *Parallel Logic Programming Techniques*, Prentice-Hall, Englewood Cliffs, N.J., 1989.

[31] Thinking Machines Corporation, *CM* FORTRAN *Reference Manual*, Cambridge, Mass., 1989.

[32] von Eicken, T., Culler, D., Goldstein, S., and Schauser, K., Active messages: A mechanism for integrating communication and computation, *Proc. 19th Intl Symp. Computer Architecture*, ACM, 1992.

[33] Young, M., et al., The duality of memory and communication in Mach, *Proc. 11th Symp. on Operating System Principles*, ACM, 63–76, 1987.

Mathematics and Computer Science Division
Building 221
Argonne National Laboratory
Argonne, Illinois 60439-4844

Recent Preprints:

James V. Burke and Jorge J. Moré, "Exposing Constraints," MCS-P308-0592.

J. N. Hagstrom, R. Hagstrom, R. Overbeek, M. Price, and L. Schrage, "Maximum Likelihood Genetic Sequence Reconstruction from Oligo Content," MCS-P309-0592.

Richard S. Varga and Amos J. Carpenter, "Some Numerical Results on Best Uniform Rational Approximation of x^α on [0,1]," MCS-P310-0592.

George Corliss, Tom Robey, Christian Bischof, Andreas Griewank, and Steve Wright, "Automatic Differentiation for PDEs -- Unsaturated Flow Case Study," MCS-P311-0692.

Christian H. Bischof and Xiaobai Sun, "A Framework for Symmetric Band Reduction and Tridiagonalization," MCS-P312-0692.

Mark T. Jones and Paul E. Plassmann, "Solution of Large, Sparse Systems of Linear Equations in Massively Parallel Applications," MCS-P313-0692.

Mark T. Jones and Paul E. Plassmann, "The Efficient Parallel Iterative Solution of Large Sparse Linear Systems," MCS-P314-0692.

Larry Wos, "Automated Reasoning Answers Open Questions," MCS-P315-0792.

R. D. C. Monteior and S. J. Wright, "A Globally and Superlinearly Convergent Potential Reduction Interior Point Method for Convex Programming," MCS-P316-0792.

Christian Bischof and Andreas Griewank, "ADIFOR: A FORTRAN System for Portable Automatic Differentiation," MCS-P317-0792.

Kevin W. Hopkins, "An Informal Introduction to Program Transformation and Parallel Processors," MCS-P318-0892.

Kevin W. Hopkins, "An Informal Introduction to Program Transformation," MCS-P319-0892.

Kevin W. Hopkins, "An Informal Introduction to Parallel Processors," MCS-P320-0892.

Man Kam Kwong, "Domain Decomposition: A Blowup Problem and the Ginzburg-Landau Equations," MCS-P321-0892.

William D. Gropp and David E. Keyes, "Domain Decomposition as a Mechanism for Using Asymptotic Methods," MCS-P322-0892.

R. M. M. Mattheij and S. J. Wright, "Parallel Stable Compactification for ODE with Parameters and Multipoint conditions," MCS-P323-0992.

Ian Foster and John Michalakes, "Massively Parallel Implementation of the Penn State/NCAR Mesoscale Model," MCS-P324-0992.

R. Hagstrom, G. S. Michaels, R. Overbeek, M. Price, and R. Taylor, "Overview of the Integrated Genomic Data System (IGD)," MCS-P325-0992.

R. G. Carter, "Fast Numerical Determination of Symmetric Sparsity Patterns," MCS-P326-0992.

A. Cherry, M. W. Henderson, W. K. Nickless, R. Olson, and G. Rackow, "Pass or Fail: A New Test for Password Legitimacy, MCS-P328-1092.