Programming in Fortran M

Ian Foster
Robert Olson
Steve Tuecke

CRPC-TR93355
October, 1993

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Preface

Fortran M is a joint development of Argonne National Laboratory and the California Institute of Technology (Caltech). Mani Chandy and his colleagues at Caltech have contributed in numerous ways. We are grateful to the many Fortran M users who have provided valuable feedback on earlier versions of this software, notably Donald Dabdub, Rajit Manohar, Berna Massingill, Sharif Rahman, John Thayer, and Ming Xu, and to Andrew Lavery for his contributions to the development of the Fortran M compiler.

Contents

Programming in Fortran M

Ian Foster, Robert Olson, and Steven Tuecke

Abstract

Fortran M is a small set of extensions to Fortran that supports a *modular* approach to the construction of sequential and parallel programs. Fortran M programs use *channels* to plug together *processes* which may be written in Fortran M or Fortran 77. Processes communicate by sending and receiving messages on channels. Channels and processes can be created dynamically, but programs remain deterministic unless specialized nondeterministic constructs are used. Fortran M programs can execute on a range of sequential, parallel, and networked computers. This report incorporates both a tutorial introduction to Fortran M and a users guide for the Fortran M compiler developed at Argonne National Laboratory.

The Fortran M compiler, supporting software, and documentation are made available free of charge by Argonne National Laboratory, but are protected by a copyright which places certain restrictions on how they may be redistributed. See the software for details. The latest version of both the compiler and this manual can be obtained by anonymous ftp from Argonne National Laboratory in the directory `pub/fortran-m` at `info.mcs.anl.gov`.

Part I

Tutorial

1 Introduction

This report provides a tutorial introduction to Fortran M and describes how to compile and run programs using Version 1.0 of the Fortran M compiler. We assume familiarity with Fortran 77.

The report is divided into three parts. The first comprises § 1–5, and provides a tutorial introduction to both the language and compiler. The second comprises § 6–8 and provides reference material on such topics as building makefiles, tuning programs, and running programs on networks. Finally, the Appendices provide a language definition and list keywords, supported machines, known deficiencies, and future plans.

1.1 About Fortran M

Fortran M is a small set of extensions to Fortran that supports a modular approach to parallel programming, permits the writing of provably deterministic parallel programs, allows the specification of dynamic process and communication structures, provides for the integration of task and data parallelism, and enables compiler optimizations aimed at communication as well as computation. Fortran M provides constructs for creating tasks and channels, for sending messages on channels, for mapping tasks and data to processors, and so on.

Because Fortran M extends Fortran 77, any valid Fortran program is also a valid Fortran M program. (There is one exception to this rule: the keyword `COMMON` must be renamed to `PROCESS COMMON`. However, this requirement can be overridden by a compiler argument; see §4.1.) The extensions themselves have a Fortran "look and feel" and are intended to be easy to use: they can be mastered in a few hours.

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 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.

Current application efforts include coupled climate models, multidisciplinary design, air quality modeling, particle-in-cell codes, and computational biology.

1.2 About the Fortran M Compiler

This report describes Version 1.0 of the Fortran M compiler. This is a preprocessor that translates Fortran M programs into Fortran 77 plus calls to a run-time communication and process management library. The Fortran 77 generated by the preprocessor is compiled with a conventional Fortran 77 compiler. Version 1.0 is a complete implementation of Fortran M, except where noted otherwise in Appendix E. See Appendix C for information on supported machines.

The communication code generated by the Fortran M compiler has yet to be optimized. However, performance studies show that it already compares favorably with p4 and PVM, two popular message-passing libraries. A deficiency of Version 1.0 is that process creation and process switching are both relatively expensive operations. This has an impact on the classes of algorithms that can be implemented efficiently in Fortran M. We expect both communication and process management performance to improve significantly in subsequent releases.

1.3 About the Fortran M Project

The Fortran M project is a joint activity of Argonne National Laboratory and the California Institute of Technology; the Fortran M compiler was developed at Argonne National Laboratory. We are continuing to develop and refine the Fortran M language and compiler. We outline some of our plans in Appendix F. We welcome comments on both the current software and development priorities.

The Fortran M mailing list is used to announce new compiler releases. Send electronic mail to `fortran-m@mcs.anl.gov` if you wish to be added to this list. Please send inquiries, comments, and bug reports to the same address.

1.4 Caveat

The Fortran M compiler should be considered unsupported research software. (We provide support on a best-efforts basis but make no guarantees.) The prospective user is urged to study the list of deficiencies provided in Appendix E of this manual before writing programs.

2 A First Example

We use a simple example to introduce both Fortran M and the Fortran M compiler. We assume that Fortran M is already installed on your computer. (If it is not, read the documentation provided with the Fortran M software release.)

Before you can use Fortran M, you must tell your environment where to find the compiler. (Normally, this will be `/usr/local/fortran-m`, but some systems may place the compiler in a different location.) If you are using the standard Unix C-shell (csh), you add one line to the *end* of the file `.cshrc` in your home directory. If the compiler has been installed in `/usr/local/fortran-m`, this line is

```
set path = ($path /usr/local/fortran-m/bin)
```

The environment variable `path` tells the Unix shell where to find various programs such as the Fortran M compiler. This shell command adds the directory containing the compiler to your shell's search path. You may have to log out and log in again for this to take effect.

2.1 A Simple Program

The `example1.fm` program creates two tasks, `producer` and `consumer`, and connects them with a channel. The channel is used to communicate a stream of integer values 1,...,5 from `producer` to `consumer`.

```
example1.fm

      program example1
      inport  (integer) pi
      outport (integer) po
      channel(in=pi, out=po)
      processes
         processcall producer(5, po)
         processcall consumer(pi)
      endprocesses
      end

      process producer(nummsgs, po)
      intent (in) nummsgs, po
      outport (integer) po
      integer nummsgs, i
      do i = 1, nummsgs
         send(po) i
      enddo
      endchannel(po)
      end

      process consumer(pi)
      intent (in) pi
      inport (integer) pi
      integer message, ioval
      receive(port=pi, iostat=ioval) message
      do while(ioval .eq. 0)
        print *, 'consumer received ', message
        receive(port=pi, iostat=ioval) message
      enddo
      end
```

The program comprises a main program and two process definitions. The main program declares two *port variables* `pi` and `po`. These can be used to receive (`INPORT`) and send (`OUTPORT`) integer messages, respectively. The `CHANNEL` statement creates a communication channel and initializes `pi` and `po` to be references to this channel. The process block (`PROCESSES/ENDPROCESSES`) creates two concurrent processes, passing the port variables as arguments.

The process definitions are distinguished by the `PROCESS` keyword. The `producer` process uses the `SEND` statement to add a sequence of messages to the message queue associated with the channel referenced by `po`. The `ENDCHANNEL` statement terminates this sequence. The `consumer` process uses the `RECEIVE` statement to remove messages from this message queue until termination is detected.

2.2 Compiling and Linking a Program

The Fortran M compiler, `fm`, is used to compile a Fortran M source file. The Fortran M compiler is used in a similar manner to other Unix-based Fortran compilers.

Because our program is contained in a file `example1.fm`, we type

```
fm -c example1.fm
```

This produces `example1.o`, which contains the object code for this Fortran M source file.

Next we must link the `example1.o` object file with the Fortran M run-time system and the system libraries. This is accomplished by running

```
fm -o example1 example1.o
```

As with most Fortran compilers, the `-o` flag specifies that the name of the executable produced by the linker is to be named `example1`.

For more information on compiling and linking Fortran M programs, see §4.1.

2.3 Running a Program

A Fortran M program is executed in the same way as other programs. For example, to run `example1`, you would type the following, where `%` is the Unix shell prompt:

```
% example1
consumer received 1
consumer received 2
consumer received 3
consumer received 4
consumer received 5
%
```

In this and subsequent examples of running programs, text typed by the user is written in *italic*, program output in `roman`, and the shell prompt is `%`.

The Fortran M run-time system has a number of run-time configurable parameters that can be controlled by command line arguments. In order to keep these run-time system arguments from interfering with the program's arguments, all arguments up to but not including the first `-fm` argument are passed to the program. All arguments after the `-fm` argument are passed to the run-time system. For example, suppose you run a Fortran M program as follows:

```
my_program my_arg1 my_arg2 -fm -nodes dalek
```

This causes `my_arg1` and `my_arg2` to be passed to the Fortran M program, and `-nodes` and `dalek` to the run-time system.

Run-time system parameters are discussed in more detail in §4.2. In addition, a complete list of these run-time system parameters, and a brief description of their meaning, can be obtained by using the `-h` argument, for example:

```
my_program -fm -h
```

3 The Fortran M Language

We now proceed to a more complete description of the Fortran M extensions to Fortran 77, summarized in Figure 1.

3.1 Processes and Ports

As illustrated in the program `example1.fm` (§2), a task is implemented in Fortran M as a *process*. A process, like a Fortran program, can define common data (labeled `PROCESS COMMON` to emphasize that it is local to the process) and 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`.

A process's dummy arguments (formal parameters) are a set of typed *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 feature is discussed in Section 3.8.) 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 and 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.

In the program `example1.fm` (§2), both `pi` and `po` are to be used to communicate messages comprising single integers. More complex message formats can be defined. For example, the following declarations define inports able to (1) receive messages comprising single integers, (2) arrays of `msgsize` reals (`p2`), and (3) a single integer and a real array with size specified by the integer, respectively. In the second and third declaration, the names `m` and `x` have scope local to the port declaration.

```
    inport (integer) p1
    inport (real x(msgsize)) p2
    inport (integer m, real x(m)) p3
```

Process:	`PROCESS`
	`PROCESS COMMON`
	`PROCESSCALL`
Interface:	`INPORT`
	`OUTPORT`
Control:	`PROCESSES/ENDPROCESSES`
	`PROCESSDO/ENDPROCESSDO`
Communication:	`CHANNEL`
	`MERGER`
	`SEND`
	`RECEIVE`
	`ENDCHANNEL`
	`MOVEPORT`
	`PROBE`
Argument Copying:	`INTENT`
Virtual Computer:	`PROCESSORS`
	`SUBMACHINE`
Process Placement:	`LOCATION`

Figure 1: Fortran M Extensions

The value of a port variable is initially a distinguished value `NULL`. It can be defined to be a reference to a channel by means of the `CHANNEL`, `MERGER`, `MOVEPORT`, or `RECEIVE` statements, to be defined below.

A port cannot appear in an assignment statement. The `MOVEPORT` statement is used to assign the value of one port to another. For example:

```
inport (integer) p1, p2
moveport(from=p1, to=p2)
```

This moves the port reference from `p1` to `p2`, and then invalidates the `FROM=` port (`p1`) by setting it to `NULL` so that it can no longer be used by `SEND`, `RECEIVE`, etc.

3.2 Creating Channels and Processes

A Fortran M program is constructed by using *process blocks* and *process do-loops* to create concurrently executing processes, which are then plugged together by using *channels* to connect inport/outport pairs. A channel is a first-in/first-out message queue with a single sender and a single receiver. In this way, processes with more complex behaviors are defined. These can themselves be composed with other processes, in a hierarchical fashion.

3.2.1 The `CHANNEL` Statement

A program creates a channel by executing the `CHANNEL` statement. This has the following general form.

`channel(in=`*inport*, `out=`*outport*`)`

This both creates a new channel and defines *inport* and *outport* to be references to this channel, with *inport* able to receive messages and *outport* able to send messages. The two ports must be of the same type. Optional `IOSTAT=` and `ERR=` specifiers can be used as in Fortran file input/output statements to detect error conditions. See Appendix A for a list of valid `IOSTAT` values.

3.2.2 The Process Block

A process call has the same form as a subroutine call, except that the special syntax `PROCESSCALL` is used in place of `CALL`. Process calls are placed in process blocks and process do-loops (defined below) to create concurrently executing processes. A process block has the general form

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

where $n \geq 0$, and the statements are process calls, process do-loops, and/or at most one subroutine call. Statements in a process block execute *concurrently*. A process block terminates, allowing execution to proceed to the next executable statement, when all of its constituent statements terminate.

One of the statements in a process block may be a subroutine call. This is denoted by the use of `CALL` instead of `PROCESSCALL` in the process block. The call is executed concurrently with the other processes in the block, but is executed in the current process.

If a process block includes only a single process call, then the `PROCESSES` and `ENDPROCESSES` statements can be omitted. Note, however, that since the parent process suspends until the new process completes execution, no additional concurrency is introduced.

3.2.3 The Process Do-Loop

A process do-loop creates multiple instances of the same process. It is identical in form to the do-loop, except that the keyword `PROCESSDO` is used in place of `DO` the body can include only a process do-loop or a process call, and the keyword `ENDPROCESSDO` is used in place of `ENDDO`. For example:

```
processdo i = 1, n
   processcall myprocess
endprocessdo
```

Process do-loops can be nested inside both process do-loops and process blocks. However, process blocks cannot be nested inside process do-loops.

We illustrate the use of the process do-loop in the `ring1.fm` program below. A total of `nodes` channels and processes are created, with the channels connecting the processes in a unidirectional ring.

ring1.fm

```
      program ring1
      parameter (nodes=4)
      inport  (integer) pi(nodes)
      outport (integer) po(nodes)
      do i = 1, nodes
         channel(in=pi(i), out=po(mod(i,nodes)+1))
      enddo
      processdo i = 1, nodes
         processcall ringnode(i, pi(i), po(i))
      endprocessdo
      end
```

3.3 Determinism

Process calls in a process block or process do-loop can be passed both ports and ordinary variables as arguments. It is illegal to pass the same port to two or more processes, as this would compromise determinism by allowing multiple processes to send or receive on the same channel.

Variables named as process arguments in a process block or do-loop are passed by value: that is, they are copied. In the case of arrays, the number of values copied is determined by the declaration in the called process. Values are also copied back upon termination of the process block or do-loop, in textual order. These copy operations ensure deterministic execution, even when concurrent processes update overlapping sections of arrays. Intent declarations (described in Section 3.8) can be used to prevent some of these copy operations from occurring.

The `MOVEPORT` statement invalidates (i.e., sets to `NULL`) the `FROM=` port when copying it to the `TO=` port. This prevents multiple ports from send or receiving on the same channel, again preserving determinism.

3.4 Communication

Each Fortran M process has its own address space. The only mechanism by which it 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, respectively, and can include `END=`, `ERR=`, and `IOSTAT=` specifiers to indicate how to recover from various exceptional conditions.

3.4.1 `SEND` and `ENDCHANNEL`

A process sends a message by applying the `SEND` statement to an outport; the outport declaration specifies the message format. A process can also call `ENDCHANNEL` to send an end-of-channel (EOC) message. `ENDCHANNEL` also sets the value of the port variable to `NULL`, preventing further messages from being sent on that port. The `SEND` and `ENDCHANNEL` statements are nonblocking (asynchronous): they complete immediately. When a `SEND` statement completes, you are guaranteed that the variables that were sent are no longer needed by the send, so they may be modified.

For example, in the program `example1.fm` (§2), the outport `po` is defined to allow the communication of single integers. The `producer` process makes repeated calls to `SEND` statement to send a sequence of integer messages, and then signals end-of-channel by a call to `ENDCHANNEL`.

Channels can also be used to communicate more complex messages. For example, in the following code fragment the `SEND` statement sends a message consisting of the integer `i` followed by the first 10 elements of the real array `a`.

```
outport (integer, real x(10)) po
integer i
integer a(10)
...
send(po) i, a
```

An array element name can be given as an argument to a SEND statement. If the corresponding message component is an array, then this is interpreted as a starting address, from which the required number of elements, as specified in the outport declaration, are taken in array element order. Hence, the following statement sends the ith row of the array b.

```
outport (integer, real x(10)) po
integer i
integer b(10,10)
...
send(po) i, b(1,i)
```

As in Fortran I/O statements, ERR= and IOSTAT= specifiers can be included to indicate how to recover from exceptional conditions. These have the same meaning as the equivalent Fortran I/O specifiers, with end-of-channel treated as end-of-file, and an operation on an undefined port treated as erroneous. Hence, an ERR=*label* specifier in a SEND or ENDCHANNEL statement causes execution to continue at the statement with the specified *label* if the statement is an undefined port. An IOSTAT=*intval* specifier causes the integer variable *intval* to be set to 0 upon successful execution and to an error value otherwise. See Appendix A for a complete list of valid IOSTAT values.

3.4.2 RECEIVE

A process receives a value by applying the RECEIVE statement to an inport. For example, the consumer process in example1.fm (§2) makes repeated calls to the RECEIVE statement so as to receive a sequence of integer messages, detecting end-of-channel by using the IOSTAT specifier, described in the preceding section. A RECEIVE statement is blocking (synchronous): it does not complete until data is available. Hence, the consumer process cannot "run ahead" of the producer.

Receive statements for more complex channel types must specify one variable for each value listed in the channel type. For example, the following is a receive statement corresponding to the send statement given as an example in the preceding section.

```
inport (integer, real x(10)) pi
integer i
real a(10)
...
receive(pi) i, a
```

An array element name can be given as an argument to a RECEIVE statement. If the corresponding message component is an array, then this is interpreted as a starting address and the required number of elements are stored in contiguous elements in array element order. Hence the following statement receives the ith row of the array b.

```
inport (integer, real x(10)) pi
integer i, j
real b(10,10)
...
receive(pi) j, b(1,i)
```

As in Fortran I/O statements, END=, ERR=, and IOSTAT= specifiers can be included to indicate how to recover from erroneous conditions. These have the same meaning as the equivalent Fortran I/O specifiers, with end-of-channel treated as end-of-file and an operation on an undefined port treated as erroneous. Hence, an END=*label* specifier causes execution to continue at the statement with the specified *label* upon receipt of a EOC message. See Appendix A for a list of the valid IOSTAT values.

3.5 Variable-Sized Messages

Array dimensions in a port declaration can include variables declared in the port declaration (as long as they appear to the left of the array declaration), parameters, and arguments to the process or subroutine in which the declaration occurs. (However, the symbol "*" cannot be used to specify an assumed size.) Variables declared within a port declaration have scope local to that declaration.

If an array dimension in a port declaration includes variables declared in the port declaration, then that port can be used to communicate arrays of different sizes. For example, the following code fragment sends a message comprising the integer num followed by num real values.

```
outport (integer n, real x(n)) po
integer num
real a(maxsize)
...
send(po) num, a
```

The following code fragment receives both the value num and num real values.

```
inport (integer n, real x(n)) pi
integer num
real b(maxsize)
...
receive(pi) num, b
```

3.6 Communication Examples

We further illustrate the use of Fortran M communication statements with the program ring2.fm. This program implements a "ring pipeline", in which NP processes are connected via a unidirectional ring. After NP-1 send-receive-compute cycles, each process has accumulated the value $\sum_{i=1}^{NP}$ in the variable sum.

```
ring2.fm

      program ring2
      parameter (np=4)
      inport  (integer) ins(np)
      outport (integer) outs(np)
      do i = 1, np
         channel(in=ins(i), out=outs(mod(i,np)+1))
      enddo
      processdo i = 1, np
         processcall ringnode(i, np, ins(i), outs(i))
      endprocessdo
      end

      process ringnode(me, np, in, out)
      intent (in) me, np, in, out
      integer me, np
      inport  (integer) in
      outport (integer) out
      buff = me
      sum = buff
      do i = 1, np-1
         send(out) buff
         receive(in) buff
         sum = sum + buff
      enddo
      endchannel(out)
      receive(in) buff
      print *, 'node ', me, ' has sum = ', sum
      end
```

3.7 Dynamic Channel Structures

The values of ports can be incorporated in messages, hence transferring the ability to send or receive on a channel from one process to another. A port that is to be used to communicate port values must have an appropriate type. For example, the following declaration specifies that inport pi will be used to receive integer outports.

```
inport (outport (integer)) pi
```

A receive statement applied to this port must take an integer outport as an argument. For example:

```
inport (outport (integer)) pi
outport (integer) to
...
receive(pi) to
```

We illustrate this language feature by sketching an implementation of worker and manager processes. (The techniques used to connect the manager and multiple workers used in this example are described in §3.9.1.) The worker process takes two outports as arguments. It uses the first to request tasks from a manager and the second to report the best result. When requesting a task from the manager, it creates a new channel, sends the outport, and waits for the new task to arrive on the inport. It closes the channel to the manager and terminates upon receipt of the task descriptor 0. The manager process is assumed to be responsible for handing out `numtasks` integer task descriptors. It repeatedly receives an outport from a worker and uses this to send a task descriptor. Once `numtasks` descriptors have been handed out, it responds to subsequent requests by sending “0”. It terminates when the requests channel is closed, indicating that all workers have terminated.

```
work_man.fm

      process worker(tasks, score)
      outport (outport (integer)) tasks
      outport (real) score
      inport  (integer) ti
      outport (integer) to
      real val, best
      integer task
      best = 0.0
      channel(in=ti, out=to)
      send(tasks) to
      receive(ti) task
      do while (task .gt. 0)
         val = compute(task)
         if(val .gt. best) best = val
         channel(in=ti, out=to)
         send(tasks) to
         receive(ti) task
      enddo
      endchannel(tasks)
      send(score) best
      endchannel(score)
      end

      process manager(pi)
      integer numtasks
      parameter (numtasks = 5)
      inport  (outport (integer)) pi
      outport (integer) request
      do i = 1, numtasks
         receive(pi) request
         send(request) i
         endchannel(request)
      enddo
      end
```

A SEND operation that communicates the value of a port variable also invalidates that port by setting that variable to NULL. This action is necessary for determinism: it ensures that the ability to send or receive on the associated channel is transferred from one process to another, rather than replicated. Hence, in the following code fragment the second send statement is erroneous and would be flagged as such either at compile time or run time.

```
      outport (outport (integer)) po
```

```
outport (integer) to
...
send(po) to
send(to) msg
```

3.8 Argument Passing

As noted in §3.3, variables passed as arguments in a process block or do-loop are, by default, copied when the process is called and again upon process termination. Copy operations can be avoided by declaring process arguments `INTENT(IN)` (copy in at call, but do not copy out) or `INTENT(OUT)` (copy out at termination, but do not copy in). The default behavior can be specified explicitly as `INTENT(INOUT)`. (See §E for the `INTENT` behavior of ports in this release.)

The program `intent1.fm` below demonstrates the use of `INTENT`.

```
intent1.fm

      program intent1
      integer n
      n = 10
      print *, 'main before: n = ', n
      processcall p(n)
      print *, 'main after: n = ', n
      end

      process p(n)
      integer n
      print *, 'p before: n = ', n
      n = 20
      print *, 'p after: n = ', n
      end
```

Running this program will yield:

```
% intent1
main before:  n = 10
p before:  n = 10
p after:  n = 20
main after:  n = 20
%
```

Adding the statement `intent (in) n` to process `p` gives:

```
% intent1
main before:  n = 10
p before:  n = 10
p after:  n = 20
main after:  n = 10
%
```

Changing this statement to `intent (out) n` yields:

```
% intent1
main before:  n = 10
p before:  n = 0
p after:  n = 20
main after:  n = 20
%
```

3.9 Nondeterministic Computations

Fortran M provides two statements that can be used to implement nondeterministic computations: `MERGER` and `PROBE`. A program that does not use these statements is guaranteed to be deterministic.

3.9.1 The `MERGER` Statement

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

The `MERGER` statement has the following general form.

`merger(in=`*inport*, `out=`*outport_specifier*)

This creates a new merger, defines *inport* to be able to receive messages from this merger, and defines the outports specified by the *outport_specifier* to be able to send messages on this merger. An *outport_specifier* can be a single outport, a comma-separated list of outports, or an implied do-loop. The *inport* and the outports in the *outport_specifier* must be of the same type. Optional `IOSTAT=` and `ERR=` specifiers can be used as in Fortran file input/output statements to detect error conditions. See Appendix A for a list of valid `IOSTAT` values.

The following `merger1.fm` example uses `MERGER` to create a manager/worker structure with a single manager and multiple workers. The manager and worker components have been previously defined in the `work_man.fm` program in §3.7. In this example, two mergers are used: one to connect `numwork` workers with the manager, and one to connect the workers with an `outmonitor` process.

```
merger1.fm

      program merger1
      integer numwork, i
      parameter (numwork = 10)
      inport  (real) scores_in
      outport (real) scores_out(numwork)
      inport  (outport (integer)) reqs_in
      outport (outport (integer)) reqs_out(numwork)

      merger(in=reqs_in, out=(reqs_out(i),i=1,numwork))
      merger(in=scores_in, out=(scores_out(i),i=1,numwork))

      processes
         processcall manager(reqs_in)
         processdo i = 1, numwork
            processcall worker(reqs_out(i), scores_out(i))
         endprocessdo
         processcall outmonitor(scores_in)
      endprocesses
      end
```

3.9.2 The `PROBE` Statement

A process can apply the `PROBE` statement to an inport to determine whether messages are pending on the associated channel. A `PROBE` statement has the general form

`probe(`*inport*`,empty=`*logical*`)`

A logical variable specified in the `EMPTY=`*variable* specifier is set to false if there is a message ready for receipt on the channel or if the channel has been closed (i.e., reached end-of-channel), and to true otherwise. In other words, the `EMPTY=`*variable* specifier is set to true if a `RECEIVE` on this *inport* would block, and to false if it would not.

In addition, `IOSTAT=` and `ERR=` specifiers can be included in its control list; these are as in the Fortran `INQUIRE` statement. Hence, applying a `PROBE` statement to an undefined port causes an integer value specified in an `IOSTAT` specifier to be set to a nonzero value and causes the execution to branch to a label provided in an `ERR=` specifier. See Appendix A for a list of valid `IOSTAT` values.

Knowledge about sends is presumed to take a nonzero but finite time to become known to a process probing an inport. Hence, a probe of an inport that references a nonempty channel may signal true if the channel values were only recently communicated. However, if applied repeatedly without intervening receives, PROBE will eventually signal false, and will then continue to do so until values are received.

The PROBE statement is useful when a process wishes to interrupt local computation to handle communications that arrive at some unpredictable rate. The process alternates between performing computation and probing for pending messages, and switchs to handling messages when PROBE returns false. For example, this is the behavior that is required when implementing a one-process-per-processor version of a branch-and-bound search algorithm. Each process alternates between advancing the local search and responding to requests for work from other processes:

```
do while (.true.)
   call advance_local_search
   probe(requests,EMPTY=empty)
   if(.not. empty) call hand_out_work
enddo
```

The PROBE statement can also be used to receive data that arrives in a nondeterministic fashion from several sources. For example, the following program handles messages of types $T1$ and $T2$, received on two ports, p1 and p2, respectively.

```
process handle_msgs(p1,p2)
inport (T1) p1
inport (T2) p2
...
do while(.true.)
   probe(p1,EMPTY=e1)
   if(.not. e1) then
      receive(p1) val1
      call handle_msg1(val1)
   endif
   probe(p2,EMPTY=e2)
   if(.not. e2) then
      receive(p2) val2
      call handle_msg2(val2)
   endif
enddo
```

A disadvantage of this program is that if no messages are pending, it consumes resources by repeatedly probing the two channels. This "busy waiting" strategy is acceptable if no other computation can be performed on the processor on which this process is executing. In general, however, it is preferable to use a non-busy-waiting technique. If $T1 = T2$, we can introduce a merger to combine the two message streams. The handle_msgs2 process then performs receive operations on its single inport, blocking until data is available.

```
merger(in=pi,(out=po(i),i=1,2))
processes
   processcall source1(po(1))
   processcall source2(po(2))
   processcall handle_msgs2(pi)
endprocesses
```

If $T1 \neq T2$, we can use the following technique. Each source process is augmented with an additional outport of type integer, on which it sends a distinctive message each time it sends a message. The integer outports are connected by a merger with an inport which is passed to the handle_msgs process. This process performs receive operations on the inport to determine which source process has pending messages.

```
merger(in=ni,(out=no(i),i=1,2))
channel(in=p1i,out=p1o)
channel(in=p2i,out=p2o)
processes
   processcall source1(1,p1o,no(1))
   processcall source2(2,p2o,no(2))
   processcall handle_msgs(p1i,p2i,ni)
endprocesses

process handle_msgs(p1,p2,pp)
inport (T1) p1
inport (T2) p2
inport (integer) pp
...
do while(.true.)
   receive(pp) id
   if(id .eq. 1) then
      receive(p1) val
   else
      receive(p2) val
   endif
   call handle_mesg(val)
enddo
```

3.10 Mapping

Process blocks and process 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, a complete program must also specify how processes are mapped to processors.

Fortran M incorporates specialized constructs designed specifically to support mapping. The `PROCESSORS` declaration specifies the shape and dimension of a virtual processor array in which a program is assumed to execute, 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. An important aspect of these constructs is that they *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.

3.10.1 Virtual Computers

Fortran M's 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. For consistency with Fortran concepts, a Fortran M virtual computer is an N-dimensional array, and the constructs that control the placement of processes within this array are modeled on Fortran's array manipulation constructs.

The `PROCESSORS` declaration is used to specify the shape and size of the (implicit) processor array on which a process executes. This is similar in form and function to the array `DIMENSION` statement. It has the general form PROCESSORS(I_1,...,I_n) where $n \geq 1$ and the I_j have the same form as the arguments to a `DIMENSION` statement. For example, the following declarations all describe a virtual computer with 256 processors.

```
processors(256)
processors(16,16)
processors(16,4,4)
```

The `PROCESSORS` declaration in the *main* program specifies the shape and size of the virtual processor array on which that program is to execute. The mapping of these virtual processors is specified at load time. This mapping may be achieved in different ways on different computers. Usually, there is a one-to-one mapping of virtual processors to physical processors. Sometimes, however, it can be useful to have more virtual processors than physical processors, for example, if developing a multicomputer program on one processor.

A `PROCESSORS` declaration in a *process* specifies the shape and size of the virtual processor array on which that particular process is to execute. As with a regular array passed as an argument, this processor array cannot be larger than that declared in its parent, but can be smaller or of a different shape.

3.10.2 Process Placement

The `LOCATION` annotation specifies the processor on which the annotated process is to execute. It is similar in form and function to an array reference. It has the general form `LOCATION(`I_1`, ...,`I_n`)`, where $n \geq 1$ and the I_j have the same form as the indices in an array reference. 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.

The following code fragment shows how the program `ring1.fm` (§3.2.3) might be extended to specify process placement. The `PROCESSORS` declaration indicates that this program is to execute in a virtual computer with 4 processors, while the `LOCATION` annotation placed on the process call specifies that each `ringnode` process is to execute on a separate virtual processor.

```
program ring1_with_mapping
parameter (nodes=4)
processors(nodes)
...
processdo i = 1, nodes
   processcall ringnode(i, pi(i), po(i)) location(i)
endprocessdo
end
```

The program `tree.fm` shows the a more complex use of mapping constructs. The process `tree` creates a set of $2n - 1$ (n a power of 2) processes connected in a binary tree. The mapping construct ensures that processes at the same depth in the tree execute on different processors, if $n \leq P$, where P is the number of processors.

```
tree.fm

      process tree(locn, n, toparent)
      intent (in) locn, n, toparent
      inport  (integer) li, ri
      outport (integer) lo, ro, toparent
      processors(16)
      if(n .gt. 1) then
         channel(in=li, out=lo)
         channel(in=ri, out=ro)
         processes
            processcall tree(locn,n/2,lo)
            processcall tree(locn+n/2,n/2,ro) location(locn+n/2)
            processcall reduce(li,ri,toparent)
         endprocesses
      else
         call leaf(toparent)
      endif
      end
```

3.10.3 Submachines

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(`$\mathtt{I}_1$`,...,`$\mathtt{I}_n$`)`, 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 are as specified by the `PROCESSORS` declaration in the process definition.

The `SUBMACHINE` annotation can be used to create several disjoint virtual computers, each comprising a subset of available processors. For example, in a coupled system comprising an ocean model and an 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 2(A) and can be specified as follows. We assume that the ocean and atmosphere models both incorporate a declaration `PROCESSORS(np,np)`; hence, the atmosphere model is executed in one half of a virtual computer of size $\mathtt{np} \times 2 \times \mathtt{np}$, and the ocean model in the other half.

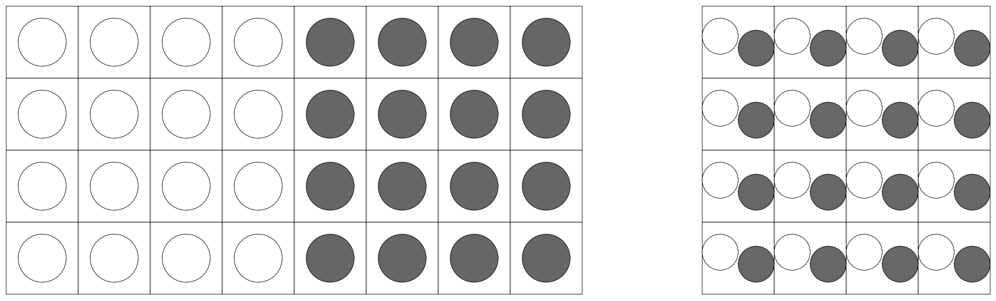

Figure 2: Alternative Mapping Strategies

```
      parameter(np=4)
      processors(np,2*np)
      ...
      processes
         processcall atmosphere(sst_in, uv_out) submachine(1,1)
         processcall ocean(sst_out, uv_in) submachine(1,np+1)
      endprocesses
```

Alternatively, it may be more efficient to map both models to the same set of processors, as illustrated in Figure 2(B). This can be achieved by changing the `PROCESSORS` declaration to `PROCESSORS(np,np)` and omitting the `SUBMACHINE` annotations. No change to the component programs is required.

4 Compiling, Running, and Debugging

The following sections provide a detailed description of the Fortran M compiler and how to use it when writing and debugging Fortran M programs.

4.1 Compiling and Linking Programs

The Fortran M compiler, `fm`, is a preprocessor rather than a true compiler. However, it is capable of compiling and linking Fortran M files (`.fm` suffix), Fortran M files with C preprocessor (CPP) directives (`.FM` suffix), Fortran files (`.f` suffix), Fortran files with CPP directives (`.F` suffix), and C files (`.c` suffix).

Every effort was made to make the Fortran M compiler conform to conventions used by most other compilers. Exceptions and additions are described in the following sections.

4.1.1 C Preprocessor

The C preprocessor (CPP) is applied to files with `.FM` and `.F` suffixes as the first stage of compilation. (For a detailed description of CPP, see any good C programming manual.) These files can contain CPP directives that specify conditional compilation, macro expansion, and constants. The following program, `cpp_ex.FM`, uses CPP directives for all of these purposes.

`cpp_ex.FM`

```
#ifndef N_NODES
#define N_NODES 1
#endif
#ifndef PRODUCER_OFFSET
#define PRODUCER_OFFSET 0
#endif
#define N_PRODUCERS (N_NODES - PRODUCER_OFFSET)

      program cpp_ex
      processors(N_NODES)
      integer n_producers
      parameter (n_producers = N_PRODUCERS)
      inport  (integer, integer) pi
      outport (integer, integer) po(n_producers)
      merger(in=pi, out=(po(i),i=1,n_producers))
      processes
#ifdef USE_CONSUMER1
         processcall consumer1(pi) location(1)
#else
         processcall consumer2(pi) location(1)
#endif
         processdo i = 1, n_producers
            processcall producer(i, po(i))
     x                        location(i+PRODUCER_OFFSET)
         endprocessdo
      endprocesses
      end
```

This program creates a single consumer process and one or more producer processes and connects the producers to the consumer by a merger. By default, all processes run on a single processor, and `consumer2` is used as the consumer process. Various aspects of this behavior can be modified at compile time through the use of `-D` compiler arguments. For example:

- Adding `-DUSE_CONSUMER1` causes `consumer1` to be used in place of `consumer2`. This sort of conditional compilation is useful when you wish to supply different versions of part of a program that will be used in different situations, such as for different machines.

- Adding `-DN_NODES=5` causes the program to create 5 producer processes and to distribute these over 5 processors, 1–5. The single consumer process runs on processor 1.
- Adding `-DN_NODES=5` and `-DPRODUCER_OFFSET=1` causes the program to create 4 producer processes and to distribute these over processors 2–4, so that the consumer process runs on a separate processor.

The result of running CPP on a `.FM` or `.F` file is a `.fm` or `.f` file, respectively, which will be passed onto the following compiler stages.

To ensure consistency across different machines, the Fortran M compiler includes its own version of CPP which it applies to files with `.FM` and `.F` suffixes. This CPP is used even if a target computer has its own CPP or if its Fortran compiler supports CPP directives. It has been our experience that different versions of CPP can differ in subtle ways, particularly when applied to Fortran programs. Please see Appendix E for information on deficiencies of the included CPP.

The behavior of CPP can be modified with the following compiler arguments:

- `-I`*path*: Add *path* to the list of paths that will be searched by CPP for files that are included through the use of `#include`.
- `-D`*def*: Add *def* as a definition during CPP.
- `-U`*def*: Remove *def* as a definition during CPP.

4.1.2 Fortran M Compiler and Linker

The Fortran M preprocessor converts a Fortran M file (`.fm`) into Fortran 77 (`.f`) and C (`__.c`) files. Fortran M statements are replaced by calls to the Fortran M libraries or to C procedures generated by the Fortran M preprocessor and located in the `__.c` file. You should need to refer to these generated `.f` and `__.c` files only when debugging, as described in §4.1.3 and §4.3.

The `.f` and `__.c` files are compiled and combined into a single object (`.o`), file.

Object files produced by the Fortran M compiler can be linked with other object files, with the Fortran M libraries, and with system libraries to produce an executable program that can be run as described in §4.2.

In addition to normal compiler arguments such as `-c`, `-o`, `-l`, and `-L` which behave as in most other compilers, and the CPP arguments described previously (§4.1.1), the behavior of the Fortran M compiler and linker can be modified with the following arguments:

- `-allow_common`: Treat each `COMMON` as if it were a `PROCESS COMMON`. (By default, Fortran M programs do not allow `COMMON` data, but instead require the use of `PROCESS COMMON` data.)

- `-rangecheck`: Compile the `.f` file with range checking turned on (if the target computer's Fortran 77 compiler supports range checking).
- `-g`: Compile and link the source files with debugging and consistency checks enabled.
- `-safe`: Compile and link the source files with consistency checks enabled.
- `-pg`: Compile and link the source files with profiling enabled.
- `-f_flag` *flag*: Pass *flag* to the Fortran compiler when compiling `.f` files.
- `-c_flag` *flag*: Pass *flag* to the C compiler when compiling `.c` files.
- `-C`: Stop after compiling the `.fm` file to a `.f` and a `__.c` file.
- `-static`: Link the executable using statically linked rather than dynamically linked libraries.

A complete list of Fortran M compiler arguments, and a brief description of their meaning, can be obtained by running `fm -h`.

4.1.3 Syntax Errors

Because the Fortran M compiler uses the C preprocessor and is itself a preprocessor, syntax errors can be detected at three stages in the compilation process:

1. The C preprocessor may detect errors in CPP directives in `.FM` or `.F` files. Line numbers in these error messages refer to the CPP (`.FM` or `.F`) file.
2. The Fortran M preprocessor may detect errors in the Fortran M code. Line numbers in these error messages refer to the Fortran M (`.fm`) file.
3. The Fortran compiler may detect errors in the `.f` file generated by the Fortran M preprocessor. Line numbers in these error messages refer to the Fortran (`.f`) file. The mapping from `.f` file errors to `.fm` or `.FM` file errors is generally fairly obvious from looking at the `.f` file.

4.2 Running Programs

The basics of running Fortran M programs are explained in §2.3. Fortran M programs are run like other programs, except that all arguments following the initial `-fm` are passed to the run-time system instead of to the user program.

Various arguments control aspects of the run-time system. Those arguments that relate to the network version of Fortran M are described in §8.5. Those that relate to debugging are described in §4.3. Another argument to the run-time system is the following:

- `-maptype` *type*: A `PROCESSORS` statement may declare more virtual processors than there are physical processors. The *type* specifies how the virtual processors are mapped to physical processors. If *type* is `cyclic` (the default), then virtual processors are mapped cyclically to physical processors. If *type* is `block`, then virtual processors are mapped blockwise to physical processors.

A complete list of the run-time system arguments, and a brief description of their meaning, can be obtained by using the `-h` argument, for example:

```
my_program -fm -h
```

4.3 Debugging Programs

Each Fortran M process is a separate Unix process, so Unix debuggers such as dbx and gdb (GNU debugger) can be used to debug Fortran M processes. Various techniques have been implemented in the Fortran M run-time system to allow debuggers to be attached to Fortran M processes in a reasonable manner.

4.3.1 Attaching a Debugger

In order to debug a Fortran M program using a debugger, that debugger must support the ability to attach to a running processes, given the process id of that process. We describe how to determine process ids below.

One such debugger is the gdb (GNU debugger). The `attach` command is used to attach gdb to a process. For example, if you wish to attach to process 4242, which is an instance of the `example1` program, you would run:

```
% gdb example1
...
(gdb) attach 4242
```

Some versions of dbx also support the ability to attach to a running process. For example, the SunOS 4.1.3 version of dbx can be attached to `example1`, process 4242 with the command:

```
dbx example1 4242
```

Consult the documentation for your debugger for more information on attaching it to a process.

4.3.2 Fatal Errors

A fatal error in a Fortran M program can be caused by an unexpected signal or a failed assertion (consistency check) in the run-time system. Normally, a fatal error will cause a message to be printed and then cause the program to be terminated.

However, the `-pause_on_fatal` run-time system flag allows you to attach to a program at the point of a fatal error. When this flag is added as a run-time system argument (after the `-fm` argument), a fatal error causes the printing of a message containing the process id of the process that has encountered the fatal error. The process then pauses instead of terminating. At this point a debugger can be attached to the process so that a postmortum analysis of the process can be conducted.

4.3.3 Pause Points

It is often useful to attach a debugger to a Fortran M process before a fatal error occurs. This can be accomplished through the use of *pause points*. A pause point is a location in a Fortran M program where a process can be paused, so that a debugger can be attached to that process.

A pause point can be added to a Fortran M program by adding the call

```
call fm_pause(id, message)
```

where *id* is an integer that identifies this pause point, and *message* is a string that will be printed at the pause point.

By default, `fm_pause` does nothing. However, adding a `-show_pausepoints` run-time system flag will cause each `fm_pause` to print a message such as

```
jayson:6036:  Pausepoint 1:  the message
```

This message says that the program has reached a pause point with an *id* of `1`, on the machine named `jayson`, in the process with process id `6036`.

The `-pause` run-time system flag will cause a process to pause at a particular pause point. For example, running

```
my_program -fm -pause 1
```

will cause a process to pause at any `fm_pause` call with an id of 1. When such a pause point is encountered, the process will print a message such as

```
jayson:6283:  Pausing at pausepoint 1:  the message
```

and will then pause. At this point a debugger can be attached to the process, using process id 6283 in this case, and debugging can commence. The debugger can be used to examine process variables, etc., and also to continue execution of the paused process.

5 Further Reading

1. I. Foster and K. M. Chandy, *Fortran M: A language for modular parallel programming*, Preprint MCS-P237-0992, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1992. This report provides the original description of the Fortran M language.

2. K. M. Chandy and I. Foster, *A deterministic notation for cooperating processes*, Preprint MCS-P346-0193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1993. This paper provides a more theoretical treatment of the Fortran M extensions to Fortran.

3. K. M. Chandy, I. Foster, C. Koelbel, K. Kennedy, and C.-W. Tseng, Integrated support for task and data parallelism, *Intl. J. Supercomputer Applications* (to appear).

Part II
Advanced Topics

6 Makefile

This section provides an example makefile for use with Fortran M programs.

makefile

```
FM = fm
FMFLAGS = -g
DEFS =

PROGS = example1 cpp_ex ring2 intent1
OTHER_OBJS = merger1.o ring1.o tree.o work_man.o

all:            $(PROGS)

other_objs:     $(OTHER_OBJS)

example1:       example1.o
        $(FM) $(FMFLAGS) $(DEFS) -o example1 example1.o

cpp_ex:         cpp_ex.o
        $(FM) $(FMFLAGS) $(DEFS) -o cpp_ex cpp_ex.o

cpp_ex.o:       cpp_ex.FM
        $(FM) $(FMFLAGS) $(DEFS) -c cpp_ex.FM

ring2:          ring2.o
        $(FM) $(FMFLAGS) $(DEFS) -o ring2 ring2.o

intent1:     intent1.o
        $(FM) $(FMFLAGS) $(DEFS) -o intent1 intent1.o

clean:
        rm -f *~ *.o *.f *__.c cpp_ex.fm *_link.c $(PROGS)

.SUFFIXES:
.SUFFIXES: .fm .o

.fm.o:
        $(FM) $(FMFLAGS) $(DEFS) -c $*.fm
```

7 Tuning Fortran M Programs

When writing Fortran M programs, you should be aware that Version 1.0 of the compiler implements the following language features efficiently:

- *Computation*: Sequential code is not modified by the Fortran M preprocessor and is compiled with conventional Fortran compilers.
- *Communication*: Preliminary experiments show that interprocessor communication rates are competitive with those achieved by message-passing libraries such as P4 and PVM.

In contrast, the following features are not implemented efficiently in Version 1.0 (but will be in future releases):

- *Process Creation*: The cost of creating a new Fortran M process is relatively high: over 100 msec on a Sun Sparcstation.
- *Port Migration*: The cost of sending a port from one process to another is relatively high: over 100 msec on Sun Sparcstation.
- *Intraprocessor Communication*: Intraprocessor (two processes on the same processor) communication performance is comparable to interprocessor communication performance on most machines. (On the SparcStation, it appears to be much *less* efficient for messages over 4k bytes.)

8 Network Specifics

The network version of Fortran M uses Berkeley stream interprocess communication (TCP sockets) to communicate between nodes. A node can run on any machine that supports TCP. Hence, a single Fortran M computation can run on several workstations of a particular type, several workstations of differing types, several processors of a multiprocessor, or a mix of workstations and multiprocessor nodes. Current restrictions are listed in §8.6.

Using network Fortran M is the same as using Fortran M on other platforms except that the user must specify on which machines Fortran M nodes are to run and may also be required to specify where on those machines the Fortran M program is to be found and the commands necessary for running Fortran M nodes on the given machines.

There are several different ways of starting network Fortran M, each appropriate for different types of network. We shall consider each of these in turn, starting with the easiest. First, we provide some background information on the Unix remote shell command `rsh`, which is used to start network Fortran M nodes.

8.1 Using `rsh`

The Unix remote shell command `rsh` is a mechanism by which a process on one machine (e.g., `host1`) can start a process on another machine (e.g., `host2`). A remote shell command can proceed only if `host1` has been given permission to start processes on `host2`. There are two ways in which this permission can be granted.

- The file `/etc/hosts.equiv` exists on `host2` and contains an entry for `host1`. This file must be created by the system administrator.
- The file `.rhosts` exists in the home directory of the user running the remote shell on `host2` and contains a line of the form

  ```
  host1 username
  ```

  where `username` is the name of the user login on `host1`. This file is created by the user.

Some sites disallow the use of `.rhosts` files for security reasons. If `.rhosts` usage is disallowed and the host machine is not in `/etc/hosts.equiv`, remote shells cannot be used to create remote processes.

The full syntax of the `rsh` command is as follows:

```
rsh hostname -l username command arguments
```

The `username` here is the login to be used on the remote machine. If `username` is not specified, it defaults to the login name of the user on the local machine. Furthermore, if the login name used on the local machine is different from the login name on the remote machine, the `.rhosts` file for the account on the remote machine must have an entry allowing access for that account on the host machine.

8.2 Specifying Nodes on the Command Line

The simplest way to start Fortran M on a network of machines is to use the `-nodes` <*nodelist*> command line argument, where *nodelist* is a colon-separated list of machine names on which Fortran M nodes are to run. For example,

```
myprogram -fm -nodes pelican:raven:plover
```

will run `myprogram` on four nodes, with one node on the machine from which this command is run and one node on each of the machines named in the *nodelist*: `pelican`, `raven`, and `plover`.

This startup method works only if

1. `rsh` (§8.1) works from the host to each machine in *nodelist*, and
2. each of the nodes shares a common filesystem with the host. The reason for this is that the initial node runs each additional node in the directory in which `myprogram` is invoked. If the initial node and an additional node have different filesystems, the `rsh` used to start up that additional node is likely to fail.

If any of these conditions does not hold, then network Fortran M must be started by using one of the alternative methods described below.

8.3 Using a Startup File

The second network Fortran M startup method that we consider can be used if nodes do not share a common file system. However, it still requires that `rsh` work from the initial node to the additional nodes.

This method uses a startup file to define the locations of remote Fortran M node processes. Lines in this file identify the machines on which nodes are to be started.

Startup File Syntax. A line of the form

command -fm $ARGS$

causes *command* to be executed. *command* is the command that invokes Fortran M on the appropriate machine. The initial process replaces *$ARGS$* at run time with the necessary arguments to Fortran M to cause it to start the node process.

Blank lines in startup files and lines starting with whitespace, `%`, or `#` are ignored.

Examples of Startup Files. A startup file containing the lines

```
rsh fulmar myprogram -fm $ARGS$
rsh plover myprogram -fm $ARGS$
```

starts one node on the machine named `fulmar`, and one node on the machine named `plover`, using the Fortran M executable called `myprogram`, resulting in a Fortran M program running on three machines.

A startup file containing the line

```
rsh fulmar -l bob myprogram -fm $ARGS$
```

starts one node using the program called `myprogram` on host `fulmar` using the Fortran M executable `myprogram` and the account for username `bob`. If we assume the initial node is being run by user `olson` on host `host-machine`, then the `.rhosts` file in the home directory of user `bob` on `fulmar` must contain the entry

```
host-machine olson
```

A startup file containing the line

```
rsh fulmar "cd /home/olson/fm; ./myprogram -fm $ARGS$"
```

runs one node on fulmar of the Fortran M executable `myprogram` after changing to the directory `/home/olson/fm`.

A startup file containing the line

```
sh -c 'echo ``cd /home/olson/fm; ./myprogram -fm $ARGS$
        < /dev/null > node.out 2>&1 &''
        | rsh fulmar /bin/sh'
```

is a more complex example that starts up one node on fulmar. This example has the desirable side effect that the `rsh` process exits after starting the Fortran M node, whereas in the other examples the `rsh` will not complete until the node process completes. Also, stdout and stderr from that node will go into the `node.out` file.

Using a Startup File. We execute network Fortran M with a startup file `fm-startup` by using the `-s` run-time system command line argument:

```
myprogram -fm -s fm-startup
```

8.4 Ending a Computation

Normally all nodes of a network Fortran M computation will exit upon completion of the computation or upon abnormal termination of any of the Fortran M processes. If for some reason this is not the case, you must log on to each machine that was executing a network Fortran M node and manually kill the Fortran M process.

8.5 Arguments to Network Version

The network version of Fortran M supports several run-time arguments to control its behavior:

- `-nodes` *node1*`:`*node2*`:`...: Start Fortran M nodes on *node1*, *node2*, etc.
- `-s` *startup-file*: Use the commands in the *startup-file* to start the Fortran M nodes.
- `-nostart`: When used in conjunction with `-nodes` or `-s`, node startup commands will be printed instead of executed. This allows nodes to be started by hand in order to, for example, be run under the control of a debugger.
- `-save_fds` *n*: Reserve *n* file descriptors for use by the user program. By default, 10 file descriptors are saved.
- `-lazy_recv`: By default, the network run-time system will receive as much data from network buffers as possible whenever a `SEND` or `RECEIVE` is done. This flag causes it to be less eager about receiving, only doing it when absolutely necessary.

8.6 Limitations of Network Version

Limits on Number of Processes. Fortran M processes are implemented as Unix processes. Hence, Unix system limits on the number of processes apply. Typically, this is in the 10–100 range per processor. However, you will likely not wish to have more than a few simultaneously active Fortran M processes on a single processor, or you (and other users on the same computer) may experience adverse effects on performance due to context switching, paging, etc.

Limits on Process Connectivity. Unix sockets are used to implement interprocess communication, with a separate socket used for each pair of processes that must communicate. Hence, Unix system limits on the number of sockets and file descriptors apply. This limit may be anywhere in the range of 50–5000 connections per process, depending on your specific version and configuration of Unix. This network version supports file descriptor caching, so the system file descriptor limit should not be a hard limit on the number of Fortran M processes. However, you might experience adverse performance effects if there are significantly more actively communicating processes than there are file descriptors. Also, the `-save_fds` runtime system argument (§8.5) can be used to reserve file descriptors for user program use.

Heterogeneous Networks. Currently, no support exists for executing Fortran M between machines with different byte orders and/or different floating-point representations. Fortran M does execute correctly between different machines if they use the same byte-ordering and floating point representation (we have run Fortran M successfully between Sun 4 and NeXT computers).

Part III

Appendices

A IOSTAT values

Many of the Fortran M calls allow the use of `IOSTAT` in order to detect error conditions. The following table lists all of the `IOSTAT` values that could be returned. Not all values apply to all Fortran M calls.

`IOSTAT`	Description
-1	End-of-file (EOF)
0	Success
1	Error
2	Inport unconnected
3	Outport unconnected
4	Inport already connected
5	Outport already connected
6	Channel already closed

B Obtaining the Fortran M Compiler

The Fortran M software is available by anonymous ftp from Argonne National Laboratory, in the `pub/fortran-m` directory on `info.mcs.anl.gov`. The latest version of this document is also available at the same location. The following session illustrates how to obtain the software in this way.

```
% ftp info.mcs.anl.gov
Connected to anagram.mcs.anl.gov.
220 anagram.mcs.anl.gov FTP server (Version 5.60+UA) ready.
Name (info.mcs.anl.gov:XXX): anonymous
331 Guest login ok, send your e-mail address as password.
Password:  /* Type your e-mail address here */
230- Guest login ok, access restrictions apply.
Argonne National Laboratory Mathematics & Computer Science Division
All transactions with this server, info.mcs.anl.gov, are logged.
230 Local time is Fri Aug 6 12:59:56 1993
ftp> cd pub/fortran-m
250 CWD command successful.
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for file list.
fm_v1.0.tar.Z
README
fm_prog_v1.0.ps.Z
fm_prog_v1.0.tar.Z
226 Transfer complete.
78 bytes received in 1.3e-05 seconds (5.9e+03 Kbytes/s)
ftp> binary
200 Type set to I.
ftp> get fm_v1.0.tar.Z
200 PORT command successful.
150 Opening BINARY mode data connection for fm_v1.0.tar.Z (XXX bytes).
226 Transfer complete.
local:  fm_v1.0.tar.Z remote:  fm_v1.0.tar.Z
XXX bytes received in YY seconds (ZZ Kbytes/s)
ftp> quit
221 Goodbye.
```

C Supported Machines

Fortran M is currently available on the following computers:

- Networks of Sun SPARCstations running SunOS version 4.1.x
- Networks of IBM RS/6000 workstations running AIX version 3.2
- Networks of SGI workstations running IRIX 4.0.5F
- Networks of NeXT workstations running NEXTSTEP 3.x
- IBM 9076 Scalable POWERparallel 1 (SP-1)

The compiler comprises a portable preprocessor and a run-time library implemented using standard Unix facilities. Hence, it should not prove difficult to port it to other computers that support TCP/IP networking.

See §B for instructions on obtaining Fortran M.

D Reserved Words

The following words may not be used as variable names or procedure names in Fortran M programs.

conn_t
fm_*
fmu_*
_u_*
yoke_*
_y_*

E Deficiencies

We are aware of the following deficiencies in Version 1.0 of the Argonne Fortran M compiler. We expect these to be remedied in subsequent releases.

In addition, see §8.6 regarding limitations in the network version of Fortran M.

1. *Fortran Syntax.* The following legal Fortran 77 statements are not processed correctly by the Fortran M compiler:

   (a) Variable declarations cannot contain expressions. (But parameters can).

   (b) The last significant (i.e. non-blank or comment) line of an `INCLUDE` file is dropped.

   (c) Inline comments (using !) are not supported.

   (d) Continuation lines must have a space in column 7.

   (e) A continuation line following a comment is considered part of the comment, not as part of the line before the comment.

   (f) `BLOCK DATA` subprograms are not handled by the FM compiler. Any required `BLOCK DATA` subprograms can be placed in a `.f` file of their own and compiled with the FM compiler.

2. *Error Messages.* The Fortran M compiler does not always generate meaningful error messages when applied to erroneous programs. If you encounter such a message, please e-mail a short example of the code that caused this error message and the message itself to "fortran-m@mcs.anl.gov".

3. Fortran M executable statements (`SEND`, `RECEIVE`, `ENDCHANNEL`, `CHANNEL`, `MERGER`, `MOVEPORT`, `PROBE`) cannot have line numbers.

4. Fortran M statements cannot be included in a logical `IF` statement.

5. In this version of Fortran M ports can only be `INTENT(IN)`, and are such by default. Ports cannot be declared `INTENT(OUT)` or `INTENT(INOUT)`.

6. Process dummy arguments that are specified as `INTENT(OUT)` should be initialized to 0. In the current implementation they are not.

7. Complex numbers cannot be sent over ports, or be passed to processes as arguments.

8. The use of a label in a `PROCESSDO` is not supported.

9. `PROCESSORS` declarations cannot contain expressions.

10. Port variables cannot be dimensioned using `0:n` syntax.

11. Dummy array arguments to processes must be dimensioned using either constants or parameters. The dimensions cannot be other dummy arguments.

12. Long file and process names can lead to truncation of Fortran M generated procedure names. In general, you should be safe if you limit process names and file base names (no path and no suffix) to less than 25 characters.

13. Procedure names cannot be passed over ports or in process calls.

14. Port variables cannot be declared to contain types with length specifiers (e.g. REAL*8).

15. `IARGC()` and `GETARG()` will return the `-fm` and following arguments as well as the user-supplied arguments.

16. *Closing channels.* You should close all outports and receive on all inports until you reach end-of-channel. Failing to do so could cause spurious error messages. For example, if a process fails to close an outport before terminating, some operations on the associated inport may cause run-time errors to be signaled.

17. `PROCESS COMMON` *definitions.* You must include any common blocks that include port variables in the process definition. (It is not sufficient to include them only in the routines that use them.) This is because the compiler uses the common block port definition to generate code to initialize any port variables in the common block. Hence, common block port definitions that are not defined in the PROCESS definition will not be properly initialized.

18. *C preprocessor.* The C preprocessor (CPP) can get confused in Fortran comments, because it does not recognize them as comments. For example, if the Fortran comment contains an unmatched single quote (i.e., the comment contains the word "it's"), CPP will take this single quote to be the beginning of a constant and proceed to take everything literally until the next single quote that it finds. All text in between those single quotes will not have CPP directive applied to them.

19. *Receiving variable length messages.* `RECEIVE` is intended to support the ability to receive messages into the middle of an array, where the specific location in which to receive is part of the message. For example, the following statement should receive 10 real values into the `i`th row of the array `b`, where `i` is defined upon receipt of the message.

    ```
    inport (integer, real x(10)) pi
    integer i
    real a(10,10)
    ...
    receive(pi) i, a(1,i)
    ```

    This does not currently work. The value of `i` that will be used in the `a(1,i)` is that value of `i` immediately before the `RECEIVE`, not the value of `i` received in the message.

F Futures Plans

This section lists new features that we expect to incorporate in future releases of the Fortran M compiler. We welcome feedback from users regarding priorities.

1. *More Robust Parser.* A more robust parser will provide full Fortran 77 support.

2. *Optimized Compiler.* An improved compiler will both remove current limitations on the number of processes and channels and improve performance. Our goal is to provide performance superior to what can be achieved conveniently using conventional message-passing libraries.

3. *Ports.* The compiler will be ported to additional parallel and distributed computer systems. Current priorities are the Intel Paragon, Thinking Machines CM-5, HP and DEC workstations, heterogeneous networks, and PCs running Windows NT.

4. *Support Tools.* We expect these to include support for replay of nondeterministic computations and a parallel profiler.

5. *Heterogeneous Applications.* An improved linker will permit different processes to execute different executables. This avoids the need to generate a single executable containing all code that may be executed by a program.

6. *Fortran 90 Features.* Fortran 90 features such as array sections will be introduced in an incremental fashion.

7. *Data Distribution Statements.* Data distribution statements similar to those defined in High Performance Fortran will be supported, allowing Fortran M programs to both define and access distributed data structures.

8. *Interfaces to Other Systems.* Interfaces will be defined to allow Fortran M programs to compose modules implemented using message-passing libraries (e.g., the MPI message passing interface standard) and data-parallel languages (e.g., High Performance Fortran).

9. *Template Libraries.* Libraries providing implementations of commonly used parallel program structures will be developed and distributed with the compiler.

G Fortran M Language Definition

This appendix is also available as Argonne technical report ANL-93/28, "Fortran M Language Definition," by Ian Foster and Mani Chandy.

G.1 Syntax

Backus-Naur form (BNF) is used to present new syntax, with nonterminal symbols in *slanted* font, terminal symbols in `TYPEWRITER` font, and symbols defined in Appendix F of the Fortran 77 standard [1] <u>`underlined`</u>. The syntax `[`*symbol*`]` is used to represent zero or more comma-separated occurrences of *symbol*; `[`*symbol*`]`$^{(1)}$ represents one or more occurrences.

G.1.1 Process, Process Block, Process Do-loop

A *process* has the same syntax as a subroutine, except that the keyword `PROCESS` is substituted for `SUBROUTINE`, `INTENT` declarations can be provided for dummy arguments, and a process cannot take an assumed size array as a dummy argument.

A *process call* can occur anywhere that a subroutine call can occur. It has the same syntax as a subroutine call, except that the keyword `PROCESSCALL` is substituted for `CALL`. In addition, process calls can occur in process blocks and process do-loops, and recursive process calls are permitted. A *process block* is a set of statements preceded by a `PROCESSES` statement and followed by a `ENDPROCESSES` statement. A block includes zero or one subroutine calls, zero or more process calls, and zero or more process do-loops. A process do-loop has the same syntax as a do-loop, except that the `PROCESSDO` keyword is used in place of `DO`, the body of the do-loop can contain only a process do-loop or a process call, and the `ENDPROCESSDO` keyword is used in place of `ENDDO`.

A port variable or port array element can be passed as an argument to only a single process in a process block or process do-loop, and then cannot be accessed in a subroutine called in that block.

G.1.2 New Declarations

Five new declaration statements are defined: `INPORT`, `OUTPORT`, `INTENT`, `PROCESSORS`, and `PROCESS COMMON`.

inport_declaration	::	`INPORT ( [`*data_type*`] ) [`*name*`]`$^{(1)}$
outport_declaration	::	`OUTPORT ( [`*data_type*`] ) [`*name*`]`$^{(1)}$
intent_declaration	::	`INTENT(IN) [`*name*`]`$^{(1)}$ |
		`INTENT(OUT) [`*name*`]`$^{(1)}$ |
		`INTENT(INOUT) [`*name*`]`$^{(1)}$
processors_declaration	::	`PROCESSORS(` *bounds* `)`
name	::	<u>`variable_name`</u> | <u>`array_name`</u> | <u>`array_declarator`</u>

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

```
data_type            ::  fortran_data_type |
                         fortran_data_type  name |
                         INPORT ( [data_type] ) |
                         OUTPORT ( [data_type] )
```

In the PROCESSORS statement, *bounds* has the same syntax as the arguments to an array_declarator. The product of the dimensions must be nonzero. Any program, process, subroutine, or function including a LOCATION or SUBMACHINE annotation must include a PROCESSORS declaration.

The symbol *fortran_data_type* denotes the six standard Fortran data types. The dimensions in an array_declarator in a port declaration can include variable declared in the port declaration, parameters, and arguments to the process or subroutine in which the declaration occurs. The symbol "*" cannot be used to specify an assumed size. Variables declared within a port declaration have scope local to that declaration.

A PROCESS COMMON statement has the same syntax as a COMMON statement.

G.1.3 New Executable Statements

There are seven new executable statements: CHANNEL, MERGER, MOVEPORT, SEND, RECEIVE, ENDCHANNEL, and PROBE. Each of these takes as arguments a list of control specifiers, termed a *control information list.* The SEND and RECEIVE statements also take other arguments. A control information list can include at most one of each specifier, except those that name ports. The number of allowable port specifiers varies from one statement to another. The first three of these statements are as follows.

```
channel_statement    ::  CHANNEL([channel_control](1))
merge_statement      ::  MERGER([merge_control](1))
moveport_statement   ::  MOVEPORT([moveport_control](1))

channel_control      ::  outport_name | OUT=outport_name |
                         inport_name | IN=inport_name |
                         IOSTAT=storage_location | ERR=label
merge_control        ::  outport_specifier | OUT=outport_specifier |
                         inport_name | IN=inport_name |
                         IOSTAT=storage_location | ERR=label
moveport_control     ::  port_name | FROM=port_name |
                         port_name | TO=port_name |
                         IOSTAT=storage_location | ERR=label

outport_specifier    ::  outport_name | data_implied_do_list
outport_name         ::  port_name
inport_name          ::  port_name
port_name            ::  variable_name | array_element_name
```

A CHANNEL statement must include two port specifiers, and these must name an outport and an inport of the same type. If the strings OUT= and IN= are omitted, these specifiers must occur as the first and second arguments, respectively.

A MERGER statement must include at least two port specifiers, and these must name an inport and one or more unique outports, all of the same type. If the strings OUT= and IN= are omitted, the outport specifiers must precede the inport specifier, which must precede any other specifiers,

In a MOVEPORT statement, the port specifiers must name two inports or two outports, both of the same type. If the strings FROM= and TO= are omitted, these specifiers must occur as the first and second arguments, respectively. The first then specifies the "from" port and the second the "to" port.

The other four statements are as follows.

send_statement :: SEND([*send_control*]$^{(1)}$) [*argument*]
receive_statement :: RECEIVE([*receive_control*]$^{(1)}$) [*variable*]
endchannel_statement:: ENDCHANNEL([*send_control*]$^{(1)}$)
probe_statement :: PROBE([*probe_control*]$^{(1)}$)

send_control :: *outport_name* | PORT=*outport_name* |
IOSTAT=*storage_location* | ERR=label
receive_control :: *inport_name* | PORT=*inport_name* |
IOSTAT=*storage_location* | ERR=label | END=label
probe_control :: *inport_name* | PORT=*inport_name* |
ERR=label | IOSTAT=*storage_location* | EMPTY=*storage_location*
storage_location :: variable_name | array_element_name

argument :: expression |
variable :: variable_name | array_element_name | array_name

If a port specifier does not include the optional characters PORT=, it must be the first item in the control information list. A *storage_location* specified in an IOSTAT= or EMPTY= specifier must have integer and logical type, respectively.

G.1.4 Mapping

The mapping annotations `LOCATION` and `SUBMACHINE` are appended to process calls:

process_call `LOCATION(`*indices*`)`
process_call `SUBMACHINE(`*indices*`)`

where *indices* has the same syntax as the arguments to an `array_element_name`.

G.1.5 Restrictions

Port variables cannot be named in `EQUIVALENCE` statements. Programs cannot include `COMMON` data; `PROCESS COMMON` must be used instead.

G.2 Concurrency

With two exceptions, a process executes sequentially, in the same manner as a Fortran program. That is, each statement terminates execution before the next is executed. The two exceptions are the process block and the process do-loop, in which statements execute *concurrently*. That is, the processes created to execute these statements may execute in any order or in parallel, subject to the constraint that any process that is not blocked (because of a `RECEIVE` applied to an empty channel) must eventually execute. A process block or process do-loop terminates, allowing execution to proceed to the next statement, when all its process and subroutine calls terminate.

A process can access its own process common data but not that of other processes. By default, process arguments are passed by value and copied back to the parent process, in textual and do-loop iteration order, upon termination of the process block or process do-loop in which the process is called, or upon termination of the process, if the process does not occur in a process block or do-loop. A dummy argument declared `INTENT(INOUT)` is treated in the same way. If a dummy argument is declared `INTENT(IN)`, then the corresponding parent argument is not updated upon termination. If a dummy argument is declared `INTENT(OUT)`, the value of the variable is defined to a default value upon entry to the process.

G.3 Channels

Processes communicate and synchronize by sending and receiving values on typed communication streams called *channels*. A channel is created by a `CHANNEL` statement, which also defines the supplied inport and outport to be references to the new channel. A channel is a first-in/first-out message queue. An element is appended to this queue by applying the `SEND` statement to the outport that references the channel. This statement is asynchronous: it returns immediately. An element is removed from the queue by applying the `RECEIVE` statement to the inport that references the channel. This statement is synchronous: it blocks until a value is available. The `ENDCHANNEL` statement appends an end-of-channel (EOC) message to the queue. The `MOVEPORT` statement copies a channel reference from one port variable to another.

These statements all take as arguments a control information list (*cilist*). The optional `IOSTAT=`, `END=`, and `ERR=` specifiers have the same meaning as the equivalent Fortran I/O specifiers, with end-of-channel treated as end-of-file, and an operation on an undefined port treated as erroneous. An implementation should also provide, as a debugging aid, the option of signaling an error if a `SEND`, `ENDCHANNEL`, or `RECEIVE` statement is applied to a port that is the only reference to a channel.

`SEND(`*cilist*`)` $\texttt{E}_1\texttt{,...,E}_n$ Add the values $\texttt{E}_1$, ..., $\texttt{E}_n$ (the sources) to the channel referenced by the outport named in *cilist* (the target). The source values must match the data types specified in the port declaration, in number and type.

`RECEIVE(`*cilist*`)` $\texttt{V}_1\texttt{,...,V}_n$ Block until the channel referenced by the inport named in *cilist* (the target) is nonempty. If the next value in the channel is not EOC, move values from the channel into the variables $\texttt{V}_1$, ..., $\texttt{V}_n$ (the destinations). The destination variables must match the data types specified in the port declaration, in number and type.

`ENDCHANNEL(`*cilist*`)` Append an EOC message to the channel referenced by the outport named in *cilist*.

`MOVEPORT(`*cilist*`)` Copy the value of the port specified "from" in *cilist* (the source) to the port specified "to" (the target), and set the source port to undefined.

A port is initially *undefined*. An undefined port becomes defined if it is included in a `CHANNEL` (or `MERGER`: see below) statement, if it occurs as a destination in a `RECEIVE`, or if it is named as the target of a `MOVEPORT` statement whose source is a defined port. Any other statement involving an undefined port is erroneous.

Application of the `ENDCHANNEL` statement to an outport causes that port to become undefined. The corresponding inport remains defined until the EOC message is received by a `RECEIVE` statement, and then becomes undefined. Both inports and outports become undefined if they are named as the source of a `SEND` or `MOVEPORT` operation.

Storage allocated for a channel is reclaimed when both (a) either the outport has been closed, or the outport goes out of scope or is redefined, and (b) either EOC is received on the inport, or the inport goes out of scope or is redefined.

G.4 Nondeterminism

The `MERGER` and `PROBE` statements are used to specify nondeterministic computations. `MERGER` is identical to `CHANNEL`, except that it can define multiple outports to be references to its message queue. Messages are added to the queue as they are sent on outports, with the order of messages from each outport being preserved and all messages eventually appearing in the queue. An EOC value is added to the queue only after it has been sent on all outports.

The PROBE statement statement is used to obtain status information for a channel. It can be applied only to an inport. The IOSTAT= and ERR= specifiers in its control list are as in the Fortran INQUIRE statement. A logical variable named in an EMPTY= specifier is assigned the value true if the channel is known to be empty, and false otherwise. Knowledge about sends is presumed to take a non-zero but finite time to become known to a process probing an inport. Hence, a PROBE of an inport that references a nonempty channel may signal true if the channel values were only recently communicated. However, if applied repeatedly without intervening receives, PROBE will eventually signal false, and will then continue to do so.

G.5 Mapping

The PROCESSORS declaration and the LOCATION and SUBMACHINE annotations have no semantic content, but determine performance by specifying how processes are to be mapped within an N-dimensional array of processors ($N \geq 1$).

The PROCESSORS declaration is analogous to a DIMENSION statement: it declares the shape and dimensions of the processor array that is to apply in the program, process, or subroutine in which it appears. As we descend a call tree, the shape of this array can change, but its size can only become smaller, not larger.

A LOCATION annotation is analogous to an array reference. It specifies the virtual processor on which the annotated process is to execute. The specified location cannot be outside the bounds of the processor array specified by the PROCESSORS declaration.

The SUBMACHINE annotation is analogous to an array reference in a subroutine call. It specifies that the annotated process is to execute in a virtual computer with its first processor specified by the annotation, and with additional processors selected in array element order. These processors cannot be outside the bounds of the processor array specified by the PROCESSORS declaration.

Index