Programming in Fortran M

Ian Foster
Robert Olson
Steven Tuecke

CRPC-TR93355
October 1993

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

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(`$I_1, \ldots, I_n$`)`, where $n \geq 0$ and the 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 np $\times$ 2 $\times$ np, and the ocean model in the other half.

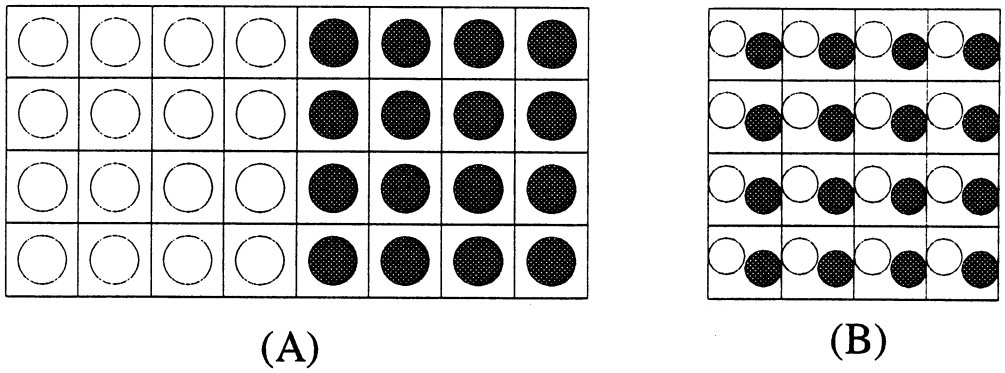
Figure 2: Alternative Mapping Strategies

```
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.