FORTRAN M:
A Language for Modular Parallel Programming

Ian T. Foster
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

K. Mani Chandy
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

June 1992

Abstract

FORTRAN M is a small set of extensions to FORTRAN 77 that supports a modular approach to the design of message-passing programs. It has the following features. (1) *Modularity.* Programs are constructed by using explicitly-declared communication channels to plug together program modules called processes. A process can encapsulate common data, subprocesses, and internal communication. (2) *Safety.* Operations on channels are restricted so as to guarantee deterministic execution, even in dynamic computations that create and delete processes and channels. Channels are typed, so a compiler can check for correct usage. (3) *Architecture Independence.* The mapping of processes to processors can be specified with respect to a virtual computer with size and shape different from that of the target computer. Mapping is specified by annotations that influence performance but not correctness. (4) *Efficiency.* FORTRAN M can be compiled efficiently for uniprocessors, shared-memory computers, distributed-memory computers, and networks of workstations. Because message passing is incorporated into the language, a compiler can optimize communication as well as computation.

1 Introduction

In the message-passing model of parallel computation, concurrently executing processes interact by exchanging messages. Originally developed for operating systems applications, the model has been widely adopted for application programming on distributed-memory computers, networks of workstations, and other parallel computer architectures. Its popularity stems from its simplicity, flexibility, and ease of implementation.

A disadvantage of the message-passing model, particularly for scientific and engineering applications, is that it does not enforce deterministic execution [27]. Hence, the programmer has no *a priori* guarantee that a program will give the same result if executed more than once with the same input. This nondeterminism is antithetical to both the scientist's need for reproducibility and ease of debugging. In addition, most message passing systems do not enforce information hiding and provide a global name space of processes. This makes it difficult to develop modular programs and reusable libraries [15].

In this paper, we describe message-passing extensions to sequential programming languages that enforce both deterministic execution and information hiding, while retaining much of the flexibility of traditional message-passing. We describe these extensions in the context of Fortran 77, and call the resulting language Fortran M. However, equivalent extensions can be defined for any sequential programming language. The extensions include constructs for defining program modules called *processes*, for specifying that processes are to execute concurrently, for establishing typed, one-to-one communication *channels* between processes, and for sending and receiving messages on channels. The resulting programming model is dynamic: processes and channels can be created and deleted dynamically, and references to channels can be included in messages.

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

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

Fortran M is supported by a theory of parallel and sequential composition of communicating processes. Key characteristics of this theory, described in a separate paper [5], include (1) proofs that a Fortran M program is deterministic even though processes and channels are created and deleted and channels are reconnected; (2) extension of sequential programming proof techniques to parallel programs; and (3) a compositional proof theory in which the specification of the whole is derived from the specifications (and not the texts) of the part.

The basic paradigm underlying FORTRAN M is *task parallelism*: the parallel execution of (possibly dissimilar) tasks. Hence, FORTRAN M complements *data-parallel* languages such as FORTRAN D [18] 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.

In the rest of this paper, we define FORTRAN M and illustrate its application to programming problems. In Sections 2 and 3, we present the constructs used to define and compose processes. In Sections 4–7, we discuss dynamic process and communication structures, nondeterministic constructs, argument passing, and process placement. Sections 8 and 9 discuss compilation and related work. We conclude in Section 10 with a discussion of future research. A language definition is provided as an appendix.

A prototype FORTRAN M compiler for sequential and parallel computers is scheduled for release in November 1992. Send electronic mail to `fortran-m@mcs.anl.gov` for details.

2 Defining Modules

In modular program design, we develop components of a program separately, as independent modules, and then combine modules to obtain a complete program [29, 10]. Interactions between modules are restricted to well-defined interfaces. Hence, module implementations can be changed without modifying other components, and the properties of a program can be determined from the specifications for its modules and the code that plugs these modules together. When successfully applied, modular design reduces program complexity and facilitates code reuse.

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

2.1 Interfaces

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

port_type (*data_type_list*) *name_list*

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

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

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

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

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

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

2.2 Communication

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

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

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

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

The ocean model might be as follows. This repeatedly sends SST data on the out-port `sst_o` and receives U and V data on the in-port `sst_i`, until EOC is detected on `sst_i`. Note the use of the `END=` specifier to indicate where execution should continue if EOC is detected.

```
      process ocean(uv_i,sst_o)
      parameter(NLAT=128, NLON=256)
C     The ports uv_i and sst_o are the external interface.
      inport (real x(NLAT,NLON), real y(NLAT,NLON)) uv_i
      outport (real x(NLAT,NLON)) sst_o
C     Process common variables.
      process common /state/ sst, u, v
      real sst(NLAT,NLON), u(NLAT,NLON), v(NLAT,NLON)
C     Repeat until EOC: send SST, recv U & V, compute SST.
      do while(.true.)
        send(sst_o) sst
        receive(uv_i,end=10) u,v
        call ocn_compute
      enddo
10    end
```

3 Composing Modules

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

3.1 Composition of Processes

A process block has the general form

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

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

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

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

3.2 Channels

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

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

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

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

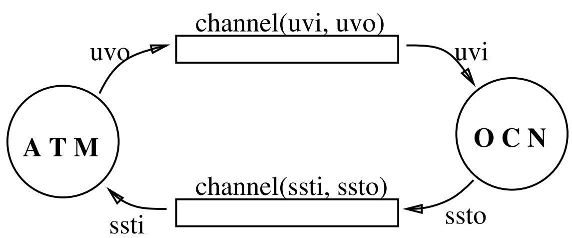


Figure 1: Coupled Ocean/Atmosphere Model

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

The value of the four port variables declared in this code fragment is initially undefined. The `CHANNEL` statements each create a channel and define their two port variable arguments to be references to this channel. These port variables are passed as arguments to the concurrently executing `atmosphere` and `ocean` processes, establishing the connections shown in Figure 1.

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

3.3 Replicating Processes

A process do-loop creates multiple instances of the same process. It is frequently used to define single program, multiple data (SPMD) computation structures, in which multiple copies of a process are connected in a regular communication structure. The process

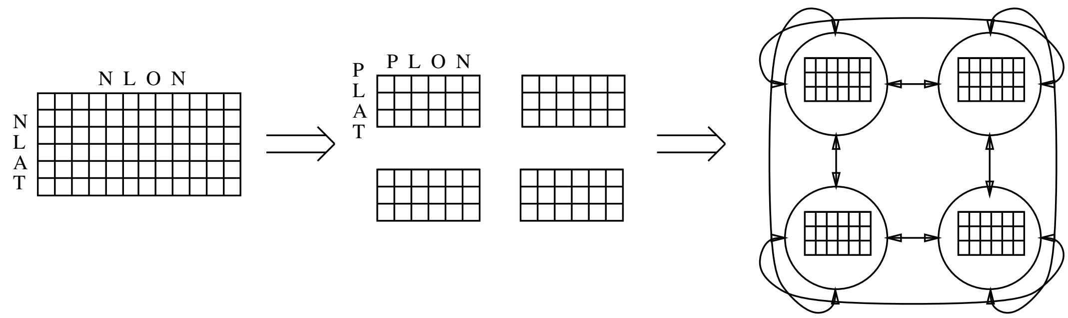


Figure 2: Parallel Atmosphere Model

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

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

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

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

Four arrays of ports, `WEi`, `WEo`, `NSi`, and `NSo`, are declared and then defined to be references to the $2\,\mathtt{NP}^2$ channels. Each `subdomain` process is passed eight of these ports; these provide in and out connections to its eight neighbors.

It is desirable to provide a parallel interface to a parallel model, so that components of the model can communicate with corresponding components of other parallel models without introducing a central bottleneck. Hence, the interface to the parallel model is also decomposed, giving two arrays of ports, `SstI` and `UvO`, each of size `NP`$\times$`NP`. Each port in these arrays is used to communicate arrays of size `PLAT`$\times$`PLON`. Each `subdomain` process is passed one element of `SstI` and one element of `UvO` as arguments.

The code used to compose the atmosphere and ocean models must be modified as follows to allow for the parallel interface. The two channels `ssto`/`ssti` and `uvo`/`uvi` are replaced with arrays of `NP`$\times$`NP` channels, and the calls to the sequential processes are replaced with calls to the parallel processes.

```
program coupled_model
parameter(NLAT=128,NLON=256,NP=16,PLON=NLON/NP,PLAT=NLAT/NP)
inport (real x(PLAT,PLON)) SstI(NP,NP)
outport (real x(PLAT,PLON)) SstO(NP,NP)
```

```
      process par_atmosphere(SstI,UvO)
      parameter(NLAT=128,NLON=256,NP=16,PLON=NLON/NP,PLAT=NLAT/NP)
C     These two port arrays define external interface.
      inport (real x(PLAT,PLON)) SstI(NP,NP)
      outport (real x(PLAT,PLON), real y(PLAT,PLON)) UvO(NP,NP)
C     Ports for communication with W & E and N & S neighbors.
      inport (real x(PLAT)) WEi(2,NP,NP)
      outport (real x(PLAT)) WEo(2,NP,NP)
      inport (real x(PLON)) NSi(2,NP,NP)
      outport (real x(PLON)) NSo(2,NP,NP)
C     Create channels used for internal communication.
      do 10 i = 1,NP
        do 11 j = 1,NP
          channel(in=NSi(2,i,j), out=NSo(1,mod(i,NP)+1,j))
          channel(out=NSo(2,i,j), in=NSi(1,mod(i,NP)+1,j))
          channel(in=WEi(2,i,j), out=WEo(1,i,mod(j,NP)+1))
          channel(out=WEo(2,i,j), in=WEi(1,i,mod(j,NP)+1))
11      continue
10    continue
C     Create NP^2 processes, with external and internal ports.
      processdo 20 i = 1,NP
        processdo 21 j = 1,NP
          call subdomain(SstI(i,j), UvO(i,j), WEi(1,i,j), WEo(1,i,j),
                         NSi(1,i,j), NSo(1,i,j))
21      continue
20    continue
      end


C     Code executed in a single subdomain.
      process subdomain(sst_i,uv_o,WEis,WEos,NSis,NSos)
      parameter(NLAT=128,NLON=256,NP=16,PLON=NLON/NP,PLAT=NLAT/NP)
C     External interface ports: for sending SST and receiving U & V.
      inport (real x(PLAT,PLON)) sst_i
      outport (real x(PLAT,PLON), real y(PLAT,PLON)) uv_o
C     Ports to and from W & E and N & S neighbors.
      inport (real x(PLAT)) WEis(2)
      outport (real x(PLAT)) WEos(2)
      inport (real x(PLON)) NSis(2)
      outport (real x(PLON)) NSos(2)
      ...
```

Program 1: Parallel Atmosphere Model

```
      inport (real x(PLAT,PLON), real y(PLAT,PLON)) UvI(NP,NP)
      outport (real x(PLAT,PLON), real y(PLAT,PLON)) UvO(NP,NP)
      ...
C     Create NP×NP channels.
      do 10 i=1,NP
        do 11 j=1,NP
          channel(out=SstO(i,j),in=SstI(i,j))
          channel(out=UvO(i,j),in=UvI(i,j))
11      continue
10    continue
      ...
C     Pass port arrays to parallel models.
      processes
        call par_atmosphere(SstI,UvO)
        call par_ocean(SstO,UvI)
      endprocesses
      end
```

3.4 Libraries

The parallel atmosphere model shows how a useful communication structure (a torus) and computational algorithm (finite differencing) can be encapsulated in a process, with a port array providing a parallel interface. Parallel implementations of other commonly used functions, such as broadcast, multicast, parallel prefix, and parallel implementations of BLAS linear algebra routines [13], can be encapsulated in the same way. As efficient implementations of these functions may require machine-specific facilities (such as hardware multicast), a Fortran M programming environment will include libraries providing high-performance implementations of these functions on different computers.

4 Dynamic Structures

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

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

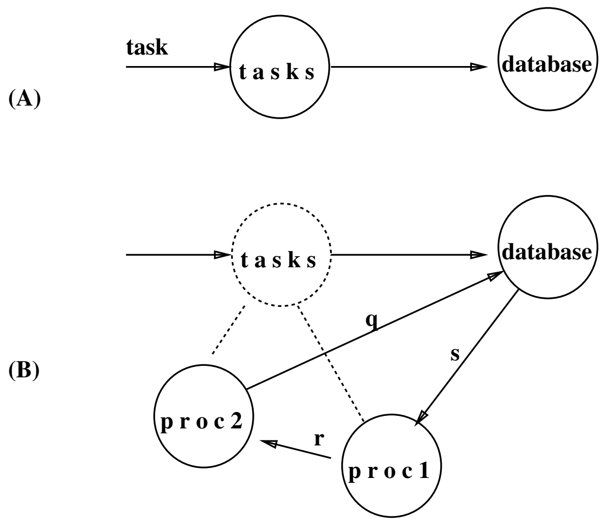


Figure 3: A Dynamic Process and Communication Structure

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

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

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

```
in_port  (integer) qi, mi, ri
out_port (in_port (integer) ) po
moveport(from=mi, to=ri)
send(po) qi
```

5 Nondeterminism

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

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

For example, consider the following problem, proposed to us by Burton Smith. `NP` `monte_carlo` processes execute independently and generate integer "scores" at irregular intervals. We wish to generate a histogram of these values. One possible solution is illustrated in Figure 4(A): we create a single `histo` process and use `MERGE` to link the out-ports of the `monte_carlo` processes and the in-port of the `histo` process. This solution can be implemented as follows. The `histo` process might be defined either to increment counts in an array or to update a histogram in a graphical display.

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

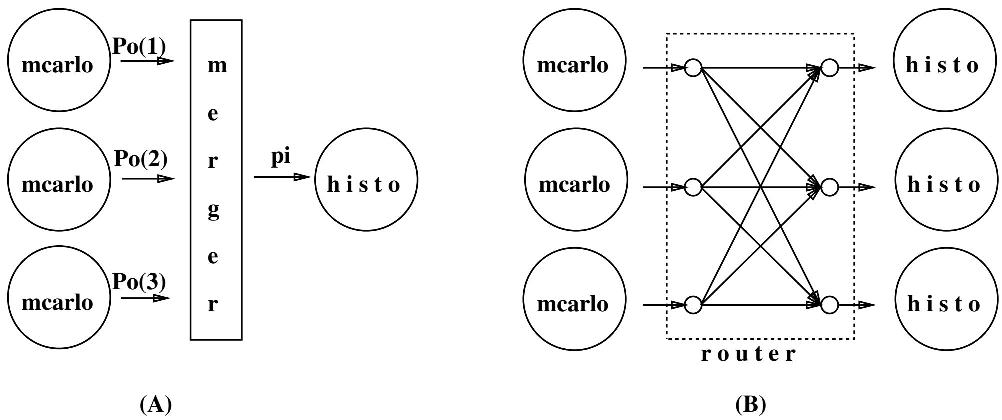


Figure 4: Histogram Problem: Centralized and Distributed Solutions

```
       call histo(pi)
       processdo 10 i = 1,NP
         call monte_carlo(Po(i))
10     continue
     endprocesses
     end
```

An alternative, less centralized solution to the problem is illustrated in Figure 4(B). The histogram is distributed among many `histo` processes, and a `router` process is used to route values to the appropriate locations. Program 2 presents a possible implementation of `router`. This creates `NP` `router_node` processes, which accept and forward addresses arriving on the in-ports `Pi`. A "crossbar" interconnect of $\mathtt{NP}^2$ channels links these processes with the out-ports `Po`; a merger combines messages routed on the `NP` channels targeted to a single out-port. This structure can route a message from `NP` inputs to `NP` outputs in constant time. If `NP` is large, the program can be modified to utilize a communication network of lower dimension, at the cost of additional communication steps.

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

6 Argument Passing

In preceding programming examples, all communication between processes has occurred via ports. For programming convenience, FORTRAN M also allows conventional argument passing between a process and the processes that it calls (its children). The values of these

```
      process router(Pi,Po)
      parameter(NP=128)
C     External interface consists of two port arrays.
      inport (integer) Pi(NP)
      outport (integer) Po(NP)
C     Ports used for internal communication.
      inport (integer) Li(NP)
      outport (integer) Lo(NP,NP)
C     Create one merger for each out-port.
      do 10 i = 1,NP
        merge(out=(Lo(i,j),j=1,NP), in=Li(i))
10    continue
C     Create NP (router_node, forward_node) pairs.
      processes
        processdo 20 i=1,NP
          call router_node(NP,Pi(i),Lo(1,i))
20      continue
        processdo 21 i=1,NP
          call forward_node(Li(i),Po(i))
21      continue
      endprocesses
      end

      process router_node(NP,pi,Lo)
      inport (integer) pi
      outport (integer) po
      inport (integer) Lo(NP)
C     Repeat: receive address and send to correct output.
      do while(.true.)
        receive(pi) iaddr
        send(Lo(mod(iaddr,NP))) iaddr/NP
      enddo
      end

      process forward_node(pi,po)
      inport (integer) pi
      outport (integer) po
C     Repeat: receive address and forward on output.
      do while(.true.)
        receive(pi) iaddr
        send(po) iaddr
      enddo
      end
```

Program 2: Router for Distributed Histogram arguments are passed to a child processes when they are created, and copied back to the parent process when the children terminate. A child process can also specify, by INTENT declarations, that particular arguments are used for input or output only. For example, the following process has three input arguments and one output argument. It computes an approximation to the integral of a function F(x) over the range $\mathtt{a.h} \leq x \leq \mathtt{b.h}$ using the rectangle rule and interval h. That is, it computes $\sum_{j=\mathtt{a}+1}^{\mathtt{b}} \mathtt{F}((j-0.5)*\mathtt{h})$.

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

A dummy argument declared INTENT(IN) cannot be modified by the process. If no intent declaration is provided for a dummy argument, or it is declared INTENT(INOUT), then the corresponding actual argument, which must be a variable, is updated after the process terminates. For a dummy argument declared INTENT(OUT), the corresponding actual argument must also be a variable, and its value is again updated upon process termination. However, in this case the value of the variable is undefined upon entry to the process.

This process is used in the following program, which computes an approximation to the integral of $F(X)$ over the interval (0,1). (For comparison, solutions to the same problem in several other parallel FORTRAN dialects are presented in [22].) The process creates NP integrate processes, each of which evaluates the integral over a specified subinterval and stores its result in an element of the array results. Upon termination of the processdo statement, elements of this array are summed by the main program.

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

A scalar value or array element can be passed to two or more processes in a process block or do-loop only if these processes all declare the corresponding dummy argument `INTENT(IN)`. For example, the integration program would be erroneous if `integrate` declared `sum` to be an array with size greater than one.

7 Process Placement

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

The techniques used to map processes to processors depends in part on the architecture of the parallel computer in question. If a small number of processors share access to a common memory, then automatic mechanisms — based, for example, on a centralized scheduler — may be effective. However, the importance of the memory hierarchy in larger parallel computers means that process placement (mapping) can be an important aspect of algorithm design. For this reason, FORTRAN M incorporates constructs that allow mapping to be specified by the programmer. These constructs *influence performance but not correctness*. Hence, we can develop a program on a uniprocessor and then tune performance on a parallel computer by changing mapping constructs.

7.1 Process Placement Constructs

The FORTRAN M process placement constructs are based on the concept of a virtual computer: a collection of virtual processors, which may or may not have the same topology as the physical computer on which a program executes [25, 36]. For consistency with FORTRAN concepts, a FORTRAN M virtual computer is an N-dimensional array, and the mapping constructs are modeled on FORTRAN 77's array manipulation constructs. The `PROCESSORS` declaration specifies the shape and dimension of a processor array, the `LOCATION` annotation maps processes to specified elements of this array, and the `SUBMACHINE` annotation specifies that a process should execute in a subset of the array [15].

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

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

A SUBMACHINE annotation is similar in form and function to an array reference passed as an argument to a subroutine. It has the general form SUBMACHINE(I_1,...,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 is as specified by the PROCESSORS declaration in the process definition.

7.2 Mapping Examples

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

```
      processors(NP,NP)
      ...
      processdo 10 i = 1,NP
        processdo 11 j = 1,NP
          call subdomain(SstI(i,j),UvO(i,j),
                         Wi(1,i,j),Wo(1,i,j),
                         Ni(1,i,j),No(1,i,j)) location(i,j)
 11     continue
 10   continue
```

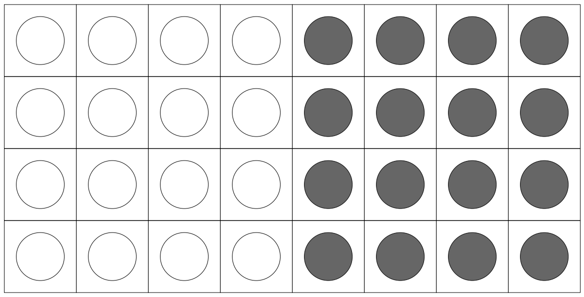
The SUBMACHINE annotation can be used to create several disjoint virtual computers, each comprising a subset of available processors. For example, in the ocean/atmosphere model, it may be desirable to execute the two models in parallel, on different parts of the same computer. This organization is illustrated in Figure 5(A) and can be specified as follows. The atmosphere model is executed in one half of a computer, and the ocean model in the other half.

```
        parameter(NP=4)
        processors(NP,2*NP)
        ...
        processes
          call atmosphere(SstI,UvO) submachine(1,1)
          call ocean(SstO,UvI)      submachine(1,NP+1)
        endprocesses
```

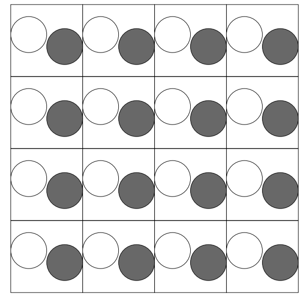
Alternatively, it may be more efficient to map both models to the same set of processors, as illustrated in Figure 5(B). This can be achieved by changing the PROCESSORS declaration to PROCESSORS(NP,NP) and omitting the SUBMACHINE annotations. No change to the component programs is required.

8 Compilation

A prototype Fortran M compiler has been developed for sequential and parallel computers and a heterogeneous network version is planned. The latter system will exploit the



(A)



(B)

Figure 5: Alternative Mapping Strategies

type information provided for channels to convert between different data representations automatically. This work will be described elsewhere when it is further advanced. We restrict ourselves here to some brief comments on performance issues.

The FORTRAN subset of FORTRAN M can be compiled with conventional compilers and thus achieves the same performance as pure FORTRAN. FORTRAN M's `SEND` and `RECEIVE` operations are translated into memory-to-memory transfers in shared-memory computers and uniprocessors and into low-level message-passing operations on distributed-memory computers. Hence, the efficiency of a simplistic distributed-memory implementation of FORTRAN M should differ little from that of equivalent programs developed with message-passing libraries. However, we also expect FORTRAN M to enable novel compiler optimizations that can significantly reduce communication and computation costs. Information about the types, contents, and sequence of messages should be obtainable by an *interprocess analysis* analogous to the interprocedural analysis performed by modern FORTRAN compilers [2, 3]; this information will allow a preprocessor to perform novel source-to-source transformations such as "process cloning", "channel merging", and "message merging". In addition, a code generator can generate specialized instruction sequences that use shared memory or drive message-passing hardware more efficiently than general purpose communication libraries. Recent research suggests that specialized communication code can improve message-passing performance by an order of magnitude [14, 31].

FORTRAN M performance also depends on the cost of process creation, scheduling, and termination operations. A preemptive scheduler is required so as to permit overlapping of computation and communication. Fortunately, these facilities are, increasingly, supported either at the operating system [38, 9] or hardware levels [21, 33, 11], or can be provided by a compiler [14].

9 Related Work

Programming notations for parallel scientific programming fall into three principal classes: coordination languages, message-passing libraries, and data parallel extensions. Here, we discuss how FORTRAN M differs from each of these approaches, focusing in particular on the issues of modularity and safety. We do not consider systems based on shared-memory models [22], as these are not easily adapted to distributed-memory machines.

In coordination language approaches, a specialized language is used to specify concurrency, communication, and synchronization; FORTRAN routines are called to perform computation. This approach has the advantage of clearly separating parallel and sequential computation, but requires the programmer to learn a new language. Coordination languages include occam [20], Strand [17], PCN [7, 16], and Delerium [24]. Delerium is purely a coordination language, while the others can be used to specify both coordination and computation. occam, derived from Hoare's Communicating Sequential Processes (CSP) [19], can specify only static computation and communication structures, does not enforce determinism, and employs synchronous communication. Strand and PCN can specify dynamic structures. Communication and synchronization are specified in terms of read and write operations on single-assignment variables, and a form of guarded command [12] is used to specify choice between alternatives. A compiler cannot in general assert that a Strand or PCN program is deterministic, because it cannot always prove that choices in guarded commands are mutually exclusive. In contrast, a FORTRAN M compiler need only verify that a program uses neither `MERGE` nor `PROBE`. Strand and PCN do not address the problem of FORTRAN common data.

In message-passing library approaches, programmers call subroutines to communicate data between processes. The number of processes is often fixed at one per physical processor. Systems such as P4 [1], Express [28], PVM [35], and Zipcode [34] provide, as primitives, an asynchronous send to a named process and a synchronous receive. The Mach operating system provides, in addition, a virtual channel construct (the port); ports can be transferred between processes in messages [38]. Mach does not restrict copying of ports, so determinism is not enforced. Libraries have the advantage of simplicity: they are language independent and do not require compiler modifications. This simplicity comes at a price, however. Compile-time checking for correct usage is not performed. As library writers know nothing about how routines will be used, they must program defensively and incorporate logic that can, in principle, be avoided in code generated by a FORTRAN M compiler. In contrast to FORTRAN M, message-passing libraries are nondeterministic and, as the name space of processes is global, do not enforce information hiding.

Related to message-passing libraries is Linda, which provides read and write operations on a shared *tuple space* [4]. Tuple space operations can emulate both message-passing communication protocols and shared data structures. Tuple space operations, like message passing, are nondeterministic and do not enforce information hiding. Actor-based message passing systems such as CE/RK [33] have some points of similarity with FORTRAN M, but are fundamentally different in that they are nondeterministic. CC++ is a shared virtual memory extension of C++ [6]. It differs from FORTRAN M in many respects, in particular its use of a shared-memory programming model.

In data parallel approaches, sequential languages are extended with directives that specify how arrays are to be decomposed and distributed over processors [37, 18, 8]. A compiler then partitions the computation using the "owner computes" rule, with each operation in the sequential program allocated to the processor containing the data that is to be operated on. This approach permits succinct specifications of parallel algorithms for regular problems and guarantees deterministic execution. When extended with support for irregular data distributions, data parallel languages can also handle some irregular problems [23, 32]. However, there are broad classes of problems for which the approach has not yet been shown to be tractable. These include highly dynamic adaptive grid problems, multidisciplinary optimization problems, and reactive systems in which a program interacts with an external environment in a nondeterministic manner. These problems can all be implemented in a straightforward manner with FORTRAN M.

10 Conclusions

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

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

FORTRAN M's extensions to FORTRAN 77 (summarized in Figure 6) can be described in a few minutes and mastered in a few hours. The extensions have a FORTRAN 77 "look and feel". For instance, the `CHANNEL`, `SEND`, `RECEIVE`, and `ENDCHANNEL` statements are similar to `OPEN`, `WRITE`, `READ`, and `ENDFILE`. Likewise, the process placement statements are modeled on FORTRAN 77 array manipulation constructs. The extensions allow programmers to develop parallel programs by plugging together modules that encapsulate both code and data. This object-oriented approach to program design supports the implementation of reusable parallel libraries and multidisciplinary applications. Furthermore, because the extensions can be implemented efficiently on a wide variety of parallel computers, application portability is achieved with little or no performance penalty. Indeed, as communication forms an integral part of the language, it should be possible to realize substantial performance improvements through compiler optimizations.

The definition of FORTRAN M opens several avenues for future research. The integration of data-parallel notations such as High Performance FORTRAN (HPF) with FORTRAN M will allow the implementation of heterogeneous applications, in which a FORTRAN M program coordinates multiple data-parallel computations. Data-parallel subroutines can be invoked in a specified processor array, with ports used for communication with FORTRAN M computations. The integration of FORTRAN 90 M constructs is also of interest. For example, array sections can be used to specify both mapping to a column of a processor array and communication of a column of a data array.

Process:	`PROCESS` `PROCESS COMMON`
Interface:	`INPORT` `OUTPORT`
Control:	`PROCESSES/ENDPROCESSES` `PROCESSDO`

Communication:	`CHANNEL`
	`MERGER`
	`SEND`
	`RECEIVE`
	`ENDCHANNEL`
	`MOVEPORT`
	`PROBE`

Performance:	`PROCESSORS`
	`LOCATION`
	`SUBMACHINE`

Figure 6: Fortran M's Extensions to Fortran 77

Acknowledgments

We are grateful to Robert Olson for his splendid efforts developing the prototype Fortran M compiler, and to John Thayer for preparing test and benchmark programs. This research was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the National Science Foundation's Center for Research in Parallel Computation, under Contract CCR-8809615.

Appendix: FORTRAN M Language Definition

A. 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 [30] `underlined`. The syntax [*symbol*] is used to represent zero or more comma-separated occurrences of *symbol*; [*symbol*]$^{(1)}$ represents one or more occurrences.

A.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. 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`, and the body of the do-loop can contain only a process do-loop or a process call.

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. Other scalar variables and array elements can be passed to two or more processes in a process block or process do-loop only if these processes all declare the corresponding dummy arguments `INTENT(IN)`. A variable or array element passed to a subroutine in a process block cannot also be passed to a process in that block.

A.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*]$^{(1)}$
machine_declaration	::	`PROCESSORS(` *bounds* `)`
name	::	`variable_name` | `array_name` | `array_declarator`
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 integer variable names in the port declaration, integer parameters, and integer arguments to the process or subroutine in which the declaration occurs. The symbol "*" cannot be used to specify an assumed size.

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

A.3 New Executable Statements

There are seven new executable statements: `CHANNEL`, `MERGE`, `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	::	`MERGE([`*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 out-port and an in-port of the same type. If the strings `OUT=` and `IN=` are omitted, these specifiers must occur as the first and second arguments, respectively.

A `MERGE` statement must include at least two port specifiers, and these must name an in-port and one or more unique out-ports, all of the same type. If the strings `OUT=` and `IN=` are omitted, the out-port specifiers must precede the in-port specifier, which must precede any other specifiers, In a `MOVEPORT` statement, the port specifiers must name two in-ports or two out-ports, 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(`[*recv_control*]$^{(1)}$`)` [*variable*]
close_statement	::	`ENDCHANNEL(`[*send_control*]$^{(1)}$`)`
probe_statement	::	`PROBE(`[*probe_control*]$^{(1)}$`)`
send_control	::	*outport_name* | `PORT=`*outport_name* |
		`IOSTAT=`*storage_location* | `ERR=`<u>`label`</u>
recv_control	::	*inport_name* | `PORT=`*inport_name* |
		`IOSTAT=`*storage_location* | `ERR=`<u>`label`</u> | `END=`<u>`label`</u>
probe_control	::	*inport_name* | `PORT=`*inport_name* |
		`ERR=`<u>`label`</u> | `IOSTAT=`*storage_location* | `EMPTY=`*storage_location*
storage_location	::	<u>`variable_name`</u> | <u>`array_element_name`</u>
argument	::	<u>`expression`</u> |
variable	::	<u>`variable_name`</u> | <u>`array_element_name`</u> | <u>`array_name`</u>

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.

A.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 <u>`array_element_name`</u>.

A.5 Restrictions

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

B. 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. A dummy argument declared `INTENT(IN)` cannot be modified by the process. If no `INTENT` declaration is provided for a dummy argument, or it is declared `INTENT(INOUT)`, then the corresponding actual argument, which must be a variable, is updated after the process terminates. If a dummy argument is declared `INTENT(OUT)`, the corresponding actual argument must also be a variable, and its value is again updated upon process termination. However, in this case the value of the variable is undefined upon entry to the process.

C. 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 in-port and out-port 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 out-port 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 in-port 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*`)` $E_1,\ldots,E_n$ Add the values E_1, ..., E_n (the sources) to the channel referenced by the out-port named in *cilist* (the target). The source values must match the data types specified in the port declaration, in number and type.

`RECEIVE(`*cilist*`)` $V_1,\ldots,V_n$ Block until the channel referenced by the in-port 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 V_1, ..., 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 out-port 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 `MERGE`: 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 out-port causes that port to become undefined. The corresponding in-port remains defined until the EOC message is received by a `RECEIVE` statement, and then becomes undefined. Both in-ports and out-ports 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 out-port has been closed, or the out-port goes out of scope, and (b) either EOC is received on the in-port, or the in-port goes out of scope.

D. Nondeterminism

The `MERGE` and `PROBE` statements are used to specify nondeterministic computations. `MERGE` is identical to `CHANNEL`, except that it can define multiple out-ports to be references to its message queue. Messages are added to the queue as they are sent on out-ports, with the order of messages from each out-port 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 out-ports.

The `PROBE` statement is used to obtain status information for a channel. Is can only be applied to an in-port. 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 in-port. Hence, a `PROBE` of an in-port 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.

E. 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 ($N \geq 0$).

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.

References

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

[2] Briggs, P., Cooper, K., Hall, M., and Torczon, L., Goal-directed interprocedural optimization, Report CRPC-TR90102, Center for Research in Parallel Computation, Rice University, Houston, Texas, 1990.

[3] Callahan, D., Cooper, K., Hood, R., Kennedy, K., and Torczon, L., Parascope: A parallel programming environment, *Intl J. Supercomputer Applications*, 2(4), 1988.

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

[5] Chandy, K. M., and Foster, I., Communicating processes, Preprint, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1992.

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

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

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

[9] Cooper, E., and Draves, R., C Threads, Technical Report, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1987.

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

[11] Dally, W. J., et al., The J-Machine: A fine-grain concurrent computer, *Information Processing 89*, G. X. Ritter (ed.), Elsevier Science Publishers B.V., North Holland, IFIP, 1989.

[12] Dijkstra, E.W., Guarded commands, nondeterminacy and the formal derivation of programs, *CACM*, 18, 453-7, 1975.

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

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

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

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

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

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

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

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

[21] Inmos, Ltd, *Transputer Reference Manual*, Prentice Hall, 1988.

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

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

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

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

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

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

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

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

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

[31] Rosing, M., and Saltz, J., Low-latency messages on distributed-memory multiprocessors, Technical Report, ICASE Report, Institute for Computer Applications in Science and Engineering, Hampton, Virginia, 1992.

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

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

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

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

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

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

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