Preprint MCS-P350-0193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1993

Chapter 1
Deterministic Parallel FORTRAN[*]

K. Mani Chandy[†] Ian Foster[‡]

Abstract

We describe FORTRAN M, message-passing extensions to FORTRAN 77 that provide deterministic execution and information hiding while preserving desirable properties of message passing.

1 Introduction

FORTRAN M is a small set of extensions to FORTRAN 77 with 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.* Programs can be compiled efficiently for vector machines, 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.

5. *Flexibility.* Programs can integrate data-parallel and control-parallel computations.

FORTRAN M is a general-purpose parallel programming language that complements data-parallel approaches such as High Performance FORTRAN. It appears particularly useful for irregular and dynamic problems, and for multidisciplinary applications that may require coordination of multiple data-parallel computations.

This paper provides an introduction to FORTRAN M. The reader is referred to [3] for a detailed language description, a discussion of compiler techniques, and a survey of related

[*]This work was supported by the National Science Foundation under Contract NSF CCR-8809615 and by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

[†]Department of Computer Science, California Institute of Technology, Pasadena, CA 91125.

[‡]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439.

Process:	`PROCESS`
	`PROCESS COMMON`
	`INPORT`
	`OUTPORT`
	`INTENT`
Concurrency:	`PROCESSES/ENDPROCESSES`
	`PROCESSDO`
Communication:	`CHANNEL`
	`MERGER`
	`SEND`
	`RECEIVE`
	`ENDCHANNEL`
	`MOVEPORT`
	`PROBE`
Mapping:	`PROCESSORS`
	`LOCATION`
	`SUBMACHINE`

Fig. 1. Fortran *M's Extensions to* Fortran *77*

work. A prototype Fortran M compiler for sequential and parallel computers is available from Argonne National Laboratory. Send electronic mail to `fortran-m@mcs.anl.gov` for details.

2 Language Overview

Fortran M's extensions to Fortran 77 are listed in Figure 1. They consist of statements for defining processes and for specifying communication, concurrent execution, nondeterministic execution, and process placement. The extensions have a Fortran 77 "look and feel." For instance, the communication statements are modeled on Fortran file I/O statements and the process placement statements on Fortran array manipulation statements.

2.1 Processes

Fortran M programs are constructed from building blocks called *processes*. 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`.

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, with `INTENT` declarations indicating whether argument values are to be copied on call and/or return. This nonessential feature is not described here.) A port variable declaration has the general form *port_type* (*data_type_list*) *name_list*

The *port_type* is either `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 define the arguments that will be passed to the subroutine.

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 optional `END=`, `ERR=`, and `IOSTAT=` specifiers to indicate how to recover from various exceptional conditions.

A process sends a value by applying the `SEND` statement to an out-port. It sends a sequence of values by repeated calls to `SEND`; it can also call `ENDCHANNEL` to send an end-of-channel (EOC) message. The `OUTPORT` declaration specifies the types of values that can be communicated on the port. 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.

2.3 Concurrent Execution

A Fortran M program is constructed by using *process blocks* and *process do-loops* to compose processes. A program creates *channels* to establish single-producer, single-consumer 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. 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*. A process block terminates, allowing execution to proceed to the next executable statement, when all its constituent statements terminate.

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.

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

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

2.4 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 connects multiple out-ports with a single in-port, to form a many-to-one communication structure. Values arriving on any out-port are copied to the in-port, with the order of messages on each out-port being preserved in the in-port and any message placed on an out-port eventually appearing on the in-port. The `probe` statement allows a process to determine whether data is pending on an in-port.

2.5 Process Placement

Fortran M incorporates constructs that allow the programmer to specify how processes are to be mapped to processors. 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.

Fortran M mapping 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 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.

3 Programming Example

We illustrate the use of Fortran M by showing how the language is used to implement a coupled ocean/atmosphere climate model. An atmosphere circulation model and an ocean circulation model 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 for simplicity that the atmosphere model is a sequential program. 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. The following code implements a model with this interface that 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 end-of-channel (EOC) message on `uv_o`. Note the use of process common to hold the `sst`, `u`, and `v` arrays. These arrays are local to the `atmosphere` process.

```
      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 /atmo/ sst, u, v
      real sst(NLAT,NLON), u(NLAT,NLON), v(NLAT,NLON)
C     Repeat TMAX times: recv SST, update U & V, send 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. Its interface is complementary to that of the atmosphere model: the in-port `uv_i` can be used to receive U and V data and the out-port `uv_o` can be used to send SST data. The body of the program repeatedly sends SST data on the out-port and receives U and V data on the in-port, until EOC is detected on `sst_i`. Note the use of the `END=` specifier in the `RECEIVE` statement to indicate where execution should continue if EOC is detected. Again, process common is used to maintain local data.

```
      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 /ocean/ sst, u, v
      real sst(NLAT,NLON), u(NLAT,NLON), v(NLAT,NLON)
C     Repeat until EOC: recv U & V, compute SST, send SST.
      do while(.true.)
        send(sst_o) sst
        receive(uv_i,end=10) u,v
        call ocn_compute
      enddo
10    end
```

The two processes are plugged together by means of two channels, one for communicating SST values and the other for communicating U and V values. This structure is illustrated in Figure 2 and is created by the following program. The program creates two channels, spawns the `atmosphere` and `ocean` processes, blocks until the process block terminates, and then terminates itself.

```
      program 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
```

We now have a complete parallel program that can be executed on a sequential or parallel computer. This program can be executed on one processor or two, or on two different computers, simply by changing mapping statements; no changes to the component modules are required. Similarly, different implementations of the ocean or atmosphere module can be substituted, as long as they use the same interface. Notice that although the execution order of the concurrently executing `atmosphere` and `ocean` processes is determined only by availability of messages on channels, the computed result does not depend on the order in which the processes execute. That is, the program is deterministic.

This example is easily extended to incorporate parallel ocean and atmosphere models [3]. The `atmosphere` and `ocean` processes use process do-loops to create multiple subprocesses

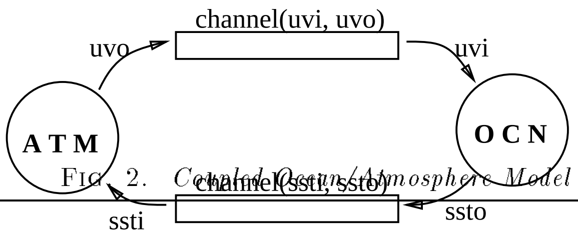

Fig. 2. *Coupled Ocean/Atmosphere Model*

and the interface between the two components is defined in terms of arrays of channels. Similarly, the example can be extended to incorporate a separate `interpolator` process that handles conversion of data between the different grid systems commonly used in ocean and atmosphere models, an I/O process that handles data reduction and file I/O, and so on. Hence, Fortran M can be used as a general-purpose modular framework for building earth system models on parallel computers [2].

4 Data Parallelism

The basic paradigm underlying Fortran M is *task parallelism*: the parallel execution of (possibly dissimilar) tasks. However, Fortran M also provides some support for data-parallel computation. Programs can use data distribution statements to create distributed arrays. Semantically, distributed arrays are indistinguishable from nondistributed arrays. That is, they are accessible only to the process in which they are declared and are copied when passed as arguments to subprocesses. Operationally, elements of a distributed array are distributed over the nodes of the virtual computer in which the process is executing. Hence, operations on a distributed array may cause communication.

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

```
processors(N)
real A(N,N), sum(N)
decomposition B(N)
align A(i,j) with B(i)
processdo i=1,N
  computesum(N, A(i,1), sum(i)) location(i)
enddo
end

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

5 Theoretical Foundations

Fortran M is supported by a theory of parallel and sequential composition of communicating processes [1]. Key characteristics of this theory 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.

6 Conclusions

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, programs that do not use two nondeterministic constructs are guaranteed to be deterministic, meaning that they produce the same output for all executions with a given input. Second, the type information required by Fortran M allows a compiler to detect many erroneous programs at compile time.

Fortran M's extensions to Fortran 77 can be described in a few minutes and mastered in a few hours. 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 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 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.

Acknowledgments

We are grateful to Bob Olson for his splendid efforts developing the prototype FORTRAN M compiler and to members of the Rice University FORTRAN compiler group for their valuable comments on the FORTRAN M language design.

References

[1] K. M. Chandy and I. Foster, *On communicating processes*, Preprint MCS-P346-0193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1993.

[2] I. Foster, FORTRAN *M as a language for building earth system models*, Preprint MCS-P345-0193, Argonne National Laboratory, and Proc. 5th ECMWF Workshop on Parallel Processing in Meteorology, ECMWF, Reading, U.K., 1992.

[3] I. Foster and K. M. Chandy, FORTRAN *M: A language for modular parallel programming*, Preprint MCS-P237-0992, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1992.

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