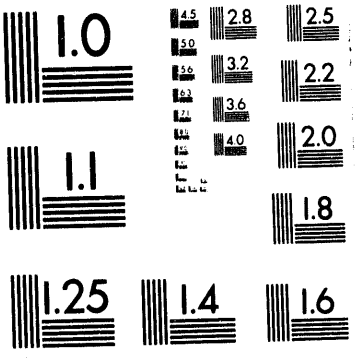
1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0

1 of 1

Parallel language constructs for paradigm integration and deterministic computations*

K. M. Chandy[a] and I. T. Foster[b]

[a]Department of Computer Science, California Institute of Technology, 256-80, Pasadena, California 91125, U.S.A.

[b]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois 60439, U.S.A.

We describe parallel extensions of sequential programming languages for writing programs that integrate different programming paradigms and that execute in heterogeneous environments comprising both distributed and shared memory. The extensions can be used to write programs with dynamic process and communication structures. Programs can use shared-memory, message passing, and data parallel programming paradigms, and can be written in a way that permits the compiler and run-time system to verify that they are deterministic. The extensions also provide the programmer with control over how data and processes are mapped to processors, and hence how computational resources are allocated to different parts of a program. A subset of these ideas has been incorporated in an extension to Fortran called Fortran M. However, the underlying sequential notation is not central to the ideas.

1. INTRODUCTION

This paper provides an overview of research at Argonne and Caltech on language extensions for parallel programming. This goal of this work is to develop programming language constructs that support dynamic process/communication structures and multi-paradigm programming but which nevertheless permit programs to be written in a way that allows the compiler and run-time system to verify that they are deterministic.

Parallel programs with dynamic process structures have computations in which processes can be created and terminate execution; communication channels can be created, reconnected, and deleted; and shared variables can be created and deleted. Programs for reactive systems, programs that use sophisticated load-balancing schemes, and programs for irregular scientific problems often have dynamic process structures. The language extensions considered in this paper are intended to support parallel programs with dynamic process structures. The extensions can also be used for computations with static process structures in which there are fixed sets of processes, channels, and shared variables.

*This research was supported by the NSF Center for Research in Parallel Computation Contract CCR-8809615 and by the DOE Office of Scientific Computing under Contract W-31-109-Eng-38.

Multiparadigm programs may combine both message passing and shared memory, and task parallelism and data parallelism. The integration of message passing and shared memory allows the use of heterogeneous networks of computers, where some nodes can be shared-memory multiprocessors. The integration of task and data parallelism is useful in multidisciplinary simulation, for example, where several disciplinary simulation models (which may be data parallel programs) must be integrated into a task parallel program. The language extensions described in this paper provide a simple mechanism for coupling multiple data-parallel programs.

Determinism is valuable for several reasons. Reasoning about deterministic programs is often simpler than reasoning about nondeterministic programs. Also, compiler support for verifying determinacy is helpful in debugging because debugging nondeterministic programs is even more intractable than debugging deterministic programs. A deterministic program will produce the same results on a single workstation or a multicomputer; this feature allows a programmer to develop a program on a workstation and later execute the program on a network of workstations or parallel computer, knowing that the output (for a given input) will remain unchanged.

Many of the ideas presented have been incorporated in an extension of Fortran called Fortran M. We chose Fortran because many people developing scientific applications use Fortran; we could equally have chosen some other sequential imperative language. An implementation of Fortran M that supports message passing and task parallelism, but not shared memory or data parallelism, is available from anonymous ftp server `info.mcs.anl.gov` (directory `/pub/fortran-m`) at Argonne National Laboratory. This compiler has been used to develop libraries of parallel programs in linear algebra, spectral methods, mesh computations, computational chemistry, and computational biology, and to explore the integration of task and data parallelism.

2. DETERMINISM

Our approach to achieving determinism is based on the *diamond property* and the Church-Rosser theorem [5, 13], well-known in functional programming.

2.1 Theory

Let G be a labeled directed graph, where each edge of the graph has a single label, and for each vertex v and each label l there is at most one edge directed from v with label l. A path in the graph is defined by the initial vertex at which the path originates, and a sequence of labels; the path is traversed by traversing the edge from the initial vertex with the first label in the sequence, then the edge with the second label, then the edge with the third label, and so on.

A *terminal* vertex is a vertex without outgoing edges. A *maximal* path is either a finite path that ends in a terminal vertex or an infinite path (i.e., a path that has an infinite number of edges).

The Diamond Property We restrict attention to graphs G with the following *diamond* property. If there are edges from a vertex v with distinct labels l and r, then there are paths l, r and r, l from v, and both paths end at the same vertex; see Figure 1. The

following theorem is proved in [5, 13].

Theorem Either all maximal paths from a vertex v are finite and end in the same terminal vertex, or all maximal paths from v are infinite.

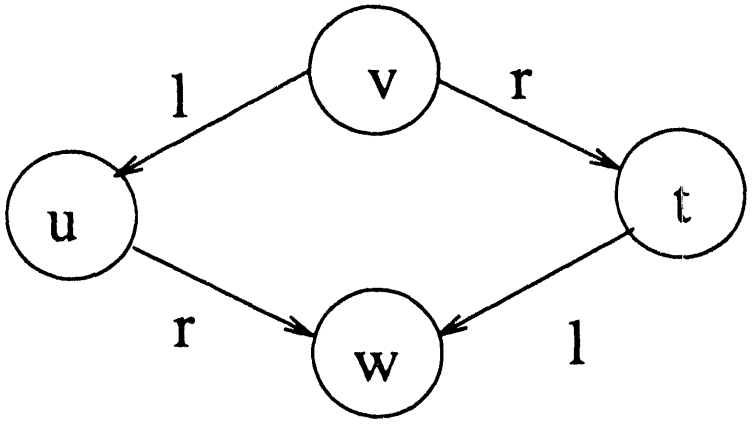

Figure 1: The Diamond Property

2.2 Application

A vertex represents a state in a parallel program, and an edge labeled r represents a state transition resulting from process r taking a step. If there is an edge labeled r from a vertex v then process r is executable in state v, and if there is no edge labeled r from vertex v then process r is suspended in state v. There is at most one edge labeled r from a vertex v because processes are deterministic, and a process does not choose nondeterministically from two or more transitions.

In terms of state transitions, the diamond property is as follows. If distinct processes l and r are both executable in a state v, and a step by process l takes the program from state v to a state u, and a step by process r takes the program from state v to a state t, then process r is executable in state u and process l is executable in state t, and the state that obtains after process r takes a step from state u is the same as the state that obtains after process l takes a step from state t.

3. PARADIGM INTEGRATION

We now explore mechanisms by which processes can communicate so that parallel programs have the diamond property, thus guaranteeing that the final state is independent of the interleaving of process computations.

3.1 Message Passing

The first communication mechanism we explore is message passing on single-writer, single-reader *channels*. Associated with each channel are two tokens: a *sender token* and a *receiver token*. An invariant of the program is: for each channel there exists at most one sender token and one receiver token.

A process can send a message on a channel if and only if it holds the sender token for that channel. Likewise, a process can receive a message from a channel if and only if it holds the receiver token for the channel. Thus the sender and receiver tokens are *capabilities* that confer certain rights to the holder of the tokens [7, 6].

The send command is nonblocking, and the receive command is blocking. The state of a channel is a queue of messages. Sending a message m on a channel appends m to the tail of the queue of messages in the channel. Receiving a message from a channel into a variable v waits until the queue of messages in the channel is nonempty, makes v become the message at the head of the queue, and then deletes the message from the queue.

Processes can communicate sender tokens and receiver tokens to other processes. Therefore different processes can send or receive messages on the same channel at different points in a computation. To ensure that the single token invariant is maintained, a token that is communicated is no longer accessible to the sending process.

The proof that parallel programs that use this (and only this) communication mechanism have the diamond property is straightforward. See Figure 2.

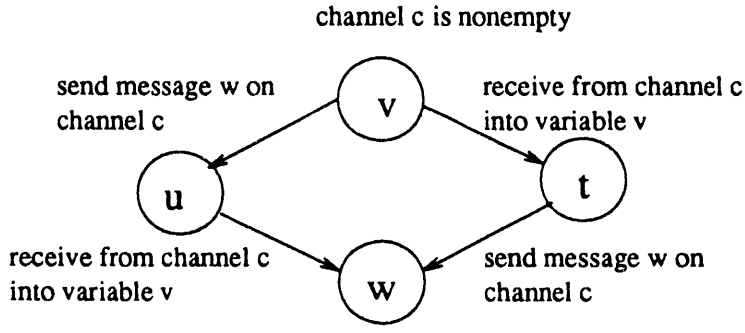

Figure 2: Channels with the Diamond Property

3.2 Shared Memory

Next we describe constructs that allow concurrent processes in parallel programs to share variables while maintaining the diamond property.

We associate with each *deterministic shared variable* a number of identical tokens. A process can write a shared variable at a point in a computation only if at that point it holds all tokens associated with the variable. A process can read a shared variable at a point in a computation only if at that point it holds at least one token associated with the variable. If a process p can write a shared variable v at a point in the computation then no other process can read or write v at that point because p holds all the tokens associated with v.

Processes can send tokens to each other. Therefore, at different points in a computation, different processes can read or write a shared variable. A process can modify the number of tokens associated with a shared variable at points in the computation at which the process holds all the tokens associated with the variable.

Programs in which processes share deterministic shared variables (and do not share any other type of variable) satisfy the diamond property because concurrent reads can

occur in arbitrary order, and no operation on a shared variable can occur concurrently with a write to the variable.

An attractive property of this restricted form of shared variable is that it can be implemented efficiently on even *weakly coherent* shared-memory architectures, and on distributed-memory architectures via message-passing.

3.3 Data Parallelism

In a data-parallel programming paradigm, the program consists of a series of operations that are applied identically to all or most elements of a data structure. Data parallel programming languages often allow the programmer to specify (a) how data is distributed over processors, and (b) the computation that is to be performed on each data item; the compiler then determines what computation and communication should be performed at each processor [15, 10].

We allow individual processes to execute programs written in a data-parallel notation. These programs may create distributed data structures which are local to that process. They may interact with other (data or task parallel) computations by operating on arrays of channels or deterministic shared variables. If the data parallel notation is deterministic and prevents replication of tokens among its threads of control (hence ensuring that the token invariant is maintained), then data-parallel processes are indistinguishable from sequential processes, and the diamond property is satisfied.

4. FORTRAN M

For concreteness, we outline how channels, deterministic shared variables, and data parallelism are incorporated into a sequential programming language (Fortran). The presentation is necessarily somewhat simplified; for details, see [8, 3].

A process declaration is syntactically identical to a subroutine except that (i) the keyword *process* replaces the keyword *subroutine*, (ii) the arguments of a process can be tokens, (iii) all process parameters other than tokens are passed by value, and (iv) the body of a process can include statements and data types, described later, that are not in the sequential language.

Processes are created by executing a *parallel block* which has the form

```
PROCESSES
   list_of_process_calls
ENDPROCESSES
```

where *list_of_process_calls* is a list of *process_call*s with end_of_line as the separator between successive elements of the list, where a *process_call* has the same syntax as a subroutine call except that the keyword PROCESSCALL is used in place of the keyword CALL. An example of a parallel block is:

```
PROCESSES
   PROCESSCALL P(V, W)
   PROCESSCALL Q(A, B, C)
   ...
ENDPROCESSES
```

where P, and Q, are process names; V, W are the arguments of P; and A, B, C are the arguments of Q.

The execution of a parallel block causes all the processes in its list of process calls to be created; the states of the newly created processes are their initial states. The processes created within the parallel block in a process t are called the children of process t. Execution of process t is suspended while any of its children are in execution, and execution of t is resumed when all its children terminate. A computation of a parallel block is a fair interleaving of the computations of its constituent processes.

All variables of a process are either local variables of the process or arguments of the process. An argument of a process can be a variable *passed by value* or it can be a token. A runtime error occurs if the same token is passed to more than one child process in a parallel block. The initial value of an uninitialized local variable is a specified default value to ensure that initial states are deterministic.

A Fortran M program is initiated as a single process executing the main program; the program terminates when this process terminates execution.

4.1 Channels

A type in the extended language is a type in the underlying sequential language or is of the form `outport(T)` or `inport(T)`, where T is a type in the extended language. The value of a variable of type `outport(T)` is either a special symbol NULL or a sender token for a channel of type T. Likewise, the value of a variable of type `inport(T)` is NULL or a receiver token for a channel of type T.

Channels are typed. A message in a channel of type T is a value of type T or a special message `end_of_channel`.

Four statements are provided in the extended language for message-passing. These statements are designed to be similar to statements in Fortran for operations on files. In the following, keywords are capitalized, variable names are italicized, *oport* is variable of type `outport(T)` and *iport* is a variable of type `inport(T)` for some T, *v* is a variable, and *ls* is a statement label.

1. `CHANNEL(OUT=`*oport*`, IN =` *iport*`)`
   This statement creates a channel of type T, and makes *oport* become the sender token associated with the channel and *iport* become the receiver token associated with the channel.

2. `SEND(PORT =` *oport*`)` *v*
   The value of *oport* is a sender token or NULL. If *oport* = NULL when the send is executed, an error occurs. If *oport* is a sender token, a message with value *v* is sent on the channel corresponding to the token. If the message itself is a token, (i.e., if the value of *v* is a token), then after the message is sent, *v* becomes NULL because the sender no longer holds the token after the token is sent.

3. `ENDCHANNEL(PORT =` *oport*`)`
   If *oport* = NULL when the statement is executed, an error occurs. If the oport is a sender token, then an `end_of_channel` message is sent on the channel corresponding to *oport* and then *oport* becomes NULL. Making *oport* NULL destroys the sender token corresponding to the channel; thus, no further messages can be sent on the channel.

4. RECEIVE(PORT = *iport*, END = *ls*) *v*
   If *iport* = NULL an error occurs. If *iport* is a receiver token, then a message is received into variable *v* from the channel corresponding to the token if the message is not **end_of_channel**. If the next message is **end_of_channel**, then *v* remains unchanged, *iport* becomes NULL (which destroys the receiver token for the channel), and execution continues from the statement labeled *ls*.

4.2 Shared variables

The syntax for declaring and using deterministic shared variables is similar to that for pointers in Fortran 90.

```
REAL, POINTER                            :: x
REAL, DETERMINISTIC_SHARED_VARIABLE :: y
```

In Fortran 90, **x** is of type pointer to a real value. Likewise, **y** is of type deterministic shared real variable.

A variable **y** of type T, DETERMINISTIC_SHARED_VARIABLE is a reference to a data structure of type T or a special symbol NULL. The *inquiry functions* TOTAL_TOKENS(y) and TOKENS_HELD(y) give the total number of tokens associated with deterministic shared variable **y**, and the number of tokens associated with **y** held by the process in which the function is called, respectively.

All operations on a deterministic shared variable **y** other than SEND, RECEIVE, and ALLOCATE, and parameter passing to processes, are operations on **y** itself, and not on the tokens associated with **y**. The operations SEND, RECEIVE, and ALLOCATE are operations on the tokens associated with **y** and do not modify the value of **y**.

Statements for dynamic storage allocation are similar to allocation statements in Fortran 90.

```
c       declare variables
        REAL, POINTER                            :: x
        REAL, DETERMINISTIC_SHARED_VARIABLE :: y

c       allocate variables
        ALLOCATE(x)
        ALLOCATE(y)
```

The first allocate statement is standard Fortran 90; it allocates storage for a new data item of type REAL and makes **x** become a pointer to it. Likewise, the second allocate statement allocates storage for a new data item **y** of type REAL DETERMINISTIC_SHARED_VARIABLE. Exactly one token is associated with a deterministic shared variable immediately after it is created. Hence TOTAL_TOKENS(y) = 1 immediately after **y** is allocated, and TOKENS_HELD(y) = 1 in the process in which the allocate statement is executed, immediately after execution of the statement. A statement is provided for increasing the number of tokens associated with a shared variable; this can only be executed by a process that holds all current tokens.

Tokens can be transferred between processes by message-passing on channels. Execution of

```
SEND(PORT = op) y(COUNT=k)
```

sends k tokens associated with variable y. Similarly, the execution of

```
RECEIVE(PORT = ip) y
```

suspends until a message arrives and receives a message from the channel corresponding to input port ip into y in the following way. Let the message received be MSG.

1. An error is posted if y is nonnull, and y and MSG reference different data items because all the tokens associated with y must reference the same data item.

2. If y is nonnull, and y and MSG reference the same data item, then the number of tokens held by the receiver corresponding to y is increased by the number of tokens in the message.

3. If y is NULL before the receive, then after the receive y references the same data item as MSG, and the number of tokens corresponding to y held by the receiver is the number of tokens in the message.

4.3 Data parallelism

Programs can use data distribution statements [10] 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 passed by value 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 require communication. Data-distribution statements allow FM programs to specify certain classes of data-parallel computations.

A complimentary approach to the integration of data parallelism is to allow Fortran M programs to call data parallel programs, written for example in Fortran D [4].

4.4. Resource management

Resource management constructs allow the programmer to specify how processes and distributed data are to be mapped to processors and hence how computational resources are to be allocated to different parts of a program [9]. These constructs influence performance but not the result computed. Hence, a program can be developed on a uniprocessor and then tuned on a parallel computer by changing only mapping constructs. For example, a programmer can specify that different components of a program are either to be executed in disjoint partitions of a multicomputer, or multiprocessed in a single partition, without changing program logic.

In Fortran M, these constructs are based on the concept of a virtual computer: a collection of virtual processors, which may or may not have the same shape as the physical computer on which a program executes. A virtual computer is an N-dimensional array, and mapping constructs are modeled on 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 [8].

5. EARLIER WORK

The work described in this paper integrates well-known ideas about the Church-Rosser theorem [5, 13], capabilities [7, 6], channels [11], and distributed shared memory [12]. An implementation in a widely-used sequential language (Fortran) provides a parallel notation that supports (i) dynamic process structures, (ii) paradigm integration and (iii) compiler verification of determinism, and that runs on multicomputer networks or (weakly-coherent) shared-memory systems. Nondeterministic constructs can be included, if required.

A comparison of Fortran M with data-parallel languages [15, 10] and high-level languages [2] highlights some of the weaknesses and strengths of our approach. Fortran M employs processes explicitly, and uses explicit exchange of tokens between processes. Care must be taken by the Fortran M programmer to avoid starvation: processes waiting for tokens that never arrive. Our experiments with writing libraries suggest that avoiding starvation is not difficult in Fortran M because if there exists *any* computation in which processes do not starve, then processes do not starve in *all* computations; so, we merely need to demonstrate *one* correct computation, and that is often easy to do by showing that the communication of tokens and messages in the Fortran M program corresponds to data flow in the sequential program. Some of this work could be handled automatically by a compiler using data-flow technology. Data-parallel languages [15, 10] and applicative languages [2] do not require the programmer to deal with processes, messages or tokens.

A weakness of Fortran M is that it is a small extension of Fortran, a sequential imperative language, whereas high-level languages such as Id and Sisal are designed from the outset to be functional. On the other hand, Fortran M uses theory from functional languages to provide a deterministic parallel extension to a language that is widely used by scientific application programmers. Since Fortran M compiles to Fortran, powerful Fortran optimizing compilers available on most platforms can be used to advantage. Furthermore, the central ideas of this paper can be used with other sequential imperative languages.

A comparison of Fortran M with parallel programs using message-passing libraries such as P4 [1] or PVM [14] is also instructive. A focus of Fortran M is the development of reliable programs by (i) separating deterministic and nondeterministic components (and allowing simpler reasoning and debugging for the deterministic parts) and (ii) type-checking messages (since channels are typed). Also, Fortran M allows dynamic process structures, and can be used to integrate shared-memory, distributed-memory and data-parallel paradigms. Libraries, by their very nature, provide no compile-time type checking and are not guaranteed to be deterministic. Also, libraries do not (generally) support dynamic process structures. Users of libraries can, however, continue to use the sequential language and compiler with which they are familiar, whereas Fortran M users have to learn the extensions to Fortran and use the Fortran M compiler. The extensions are simple, and the time required to learn the extensions is of the same order as the time required to learn a message-passing library.

References

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

[2] Cann, D. C., J. T. Feo, and T. M. DeBoni, Sisal 1,2: High Performance Applicative Computing, *Proc. Symp. Parallel and Distributed Processing*, IEEE CS Press, Los Alamitos, Calif., 1990, 612–616.

[3] Chandy, K. M. and I. Foster, A Deterministic Notation for Cooperating Processes, Preprint MCS-P346-0193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill. 60439, 1993.

[4] Chandy, K. M., I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng, Integrated Support for Task and Data Parallelism, *Intl J. Supercomputer Applications* (to appear).

[5] Church, A. and J. B. Rosser, Some Properties of Conversion, *Trans. American Math. Soc.*, 39, 1936, 472–482.

[6] Cohen, E. and D. Jefferson, Protection in the Hydra Operating System, *Proc. 5th Symp. Operating Systems Principles*, ACM, 1975, 141–150.

[7] Dennis, J. B., and E. C. Van Horn, Programming Semantics for Multiprogrammed Computations, *Comm. ACM*, 9, 1966, 143–155.

[8] Foster, I. and K. M. Chandy, Fortran M: A Language for Modular Parallel Programming, Preprint MCS-P327-0992, Argonne National Laboratory, Argonne Ill. 60439, 1992.

[9] Foster, I., R. Olson, and S. Tuecke, Productive Parallel Programming: The PCN Approach, *Scientific Programming*, 1(1), 1992.

[10] Fox, G., S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, M.Wu, Fortran D Language Specification, Technical Report TR90-141, Computer Science, Rice Univ., Houston, TX, 1990.

[11] Hoare, C. A. R., Communicating Sequential Processes, *Comm. ACM*, 21(8), 1969, 666–677.

[12] Li, K., and P. Hudak, Memory Coherence in Shared Virtual Memory Systems, *ACM Trans. Comp. Systems*, 7(4), 1989, 321–359.

[13] McLennan, B. J., *Functional Programming: Practice and Theory*, Addison-Wesley, Reading, Mass. 1990

[14] Sunderam, V., PVM: A Framework for Parallel Distributed Computing, *Concurrency Practice and Experience*, 2, 1990, 315–339.

[15] Thinking Machines Corporation, *CM Fortran Reference Manual*, Thinkin ٦ Machines, Cambridge, Mass., 1989.

DATE
FILMED
2/22/94
END