A Deterministic Notation for Cooperating Processes

K. Mani Chandy*
California Institute of Technology, 256-80
Pasadena, California 91125
mani@vlsi.cs.caltech.edu

Ian T. Foster†
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, Illinois 60439
itf@mcs.anl.gov

August 30, 1993

Abstract

This paper proposes extensions of sequential programming languages for parallel programming that have the following features:

1. **Dynamic Structures** The process structure is dynamic: Processes and variables can be created and deleted.
2. **Paradigm Integration** The programming notation allows shared memory and message passing.
3. **Determinism** Demonstrating that a program is deterministic — all executions with the same input produce the same output — is straightforward. A program can be written so that the compiler can verify that the program is deterministic. Nondeterministic constructs can be introduced in a sequence of refinement steps to obtain greater efficiency if required.

The ideas have been incorporated in an extension of Fortran, but the underlying sequential imperative language is not central to the ideas described here.

Keywords: parallel programming languages, determinism, functional programming, multicomputers, debugging

*Supported by NSF Center for Research in Parallel Computation Contract CCR-8809615

†Supported by NSF Center for Research in Parallel Computation Contract CCR-8809615 and DOE Contract W-31-109-Eng-38

1 Introduction

This paper describes a concurrent language in which the process/communication structure can be dynamic, and in which programs can be written in a way that allows the compiler and run-time system to verify that they are deterministic. The ideas in this paper are derived from the Church-Rosser theorem about systems that obey the diamond property [6, 18] and from the concept of *capabilities* in operating systems [8, 7]. The ideas have been incorporated in an extension of Fortran because many people developing scientific applications use Fortran; we could, however, have chosen some other sequential imperative language.

1.1 Dynamic Process Structures

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. In this paper we suggest extensions to sequential notations for expressing 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.

1.2 Paradigm Integration

The notation integrates message-passing and shared-memory models thus allowing for the use of heterogeneous networks of computers, where some nodes of the network can be shared-memory multiprocessors.

1.3 Determinism

Small changes in the value of a variable can cause an unstable numeric computation to diverge. Programmers are required to demonstrate that outputs of such programs are functions of their inputs, though nondeterminism in interleaved execution is acceptable. The notation presented here allows programs to be written in a way that allows the compiler and run-time system to verify that programs are deterministic.

Reasoning about deterministic programs is often simpler than reasoning about nondeterministic programs. Also, compiler support for verifying determinacy is extremely helpful in debugging because debugging nondeterministic programs is even more intractable than debugging deterministic programs.

A deterministic program will produce exactly 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.

All executions of a program with the same input produce the same output. In particular, all executions of a program with the same input must produce the same error file to aid in debugging. A runtime error causes an error message to be appended to the error file, and the statement that causes the error remains suspended while other error-free processes continue execution. The error file is ordered by the process ids in which the errors occur, and the scheme employed to determine process ids guarantees that a process gets the same id in all executions of a program with the same input.

1.4 Contribution

The contribution of this paper is to incorporate well-known ideas about the Church-Rosser theorem, capabilities, channels, distributed shared memory and single-assignment variables into a widely-used sequential language to get a parallel notation that supports dynamic process structures, paradigm integration, and compiler verification of determinism, and that runs on multicomputer networks or weakly-coherent shared-memory systems. A great deal of work has been carried out on functional (equational and applicative) languages that guarantee that the output of a program is a function of its input. See the descriptions of Id, Haskell, Sisal and Scheme in [9], for example. The theory of such languages is based, in part, on the Church-Rosser theorem [18].

A great deal of work has also been carried out on capabilities [8, 15, 14], message-passing using channels [13], shared-memory programming models on distributed-memory machines [16], and single-assignment variables [4, 11, 5].

Our contribution is to integrate the earlier work into a simple extension of Fortran 77 to allow developers of parallel scientific applications to benefit from the earlier work while using languages and tools with which they are familiar.

2 The Central Idea

First, we review the central idea of the *diamond property* and the Church-Rosser theorem [6, 18], and later use the idea to develop constructs for a parallel extension of Fortran 77.

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 intial 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.

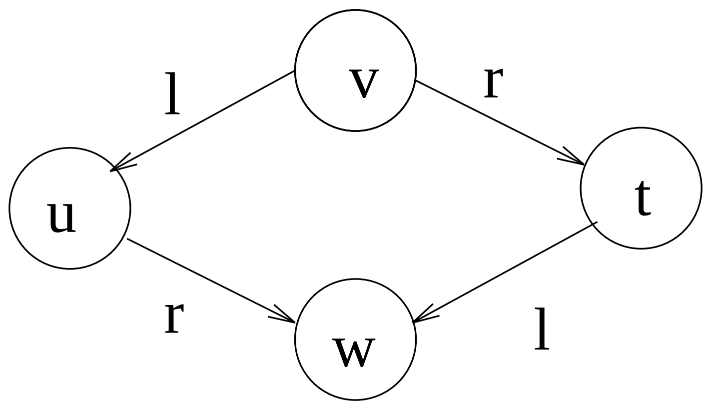

Figure 1: The Diamond Property

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.

Proof: A proof, see [6, 18], is as follows. Let P be a finite path from v that ends in a terminal state w, and let R be any maximal path from v. Let P have n edges. We construct a sequence of paths Q_k, $n \geq k \geq 0$, from v where (i) Q_k ends in state w, all k, and (ii) the first k labels of Q_k and R are the same.

The construction is by induction on k with base case $k = 0$ and $Q_0 = P$. For $k > 0$, obtain Q_k by permuting Q_{k-1} as follows. If the k-th labels of Q_k and R are the same, then $Q_k = Q_{k-1}$. Otherwise, from the diamond property and since Q_{k-1} ends in a terminal vertex, the k-th label of R appears after the k-th position in Q_{k-1}; move the first occurrence of this label after the k-th position to the k-th position, leaving the order of all other labels unchanged. It follows from the diamond property, that this permutation keeps the final state unchanged. Hence Q_k also ends in state w.

It follows from the construction, that Q_n ends in state w, and the first n labels of Q and R are the same. Since, there are no edges from w it follows that there are only n labels in R

and therefore R ends in state w.

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.

Next, we 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.

Single-Reader, Single-Writer Channels The first communication mechanism we explore is message passing on 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 one sender token and at most 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 [20].

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 send 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.

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

A bounded-buffer channel in which the sender is blocked while the channel is full (and with

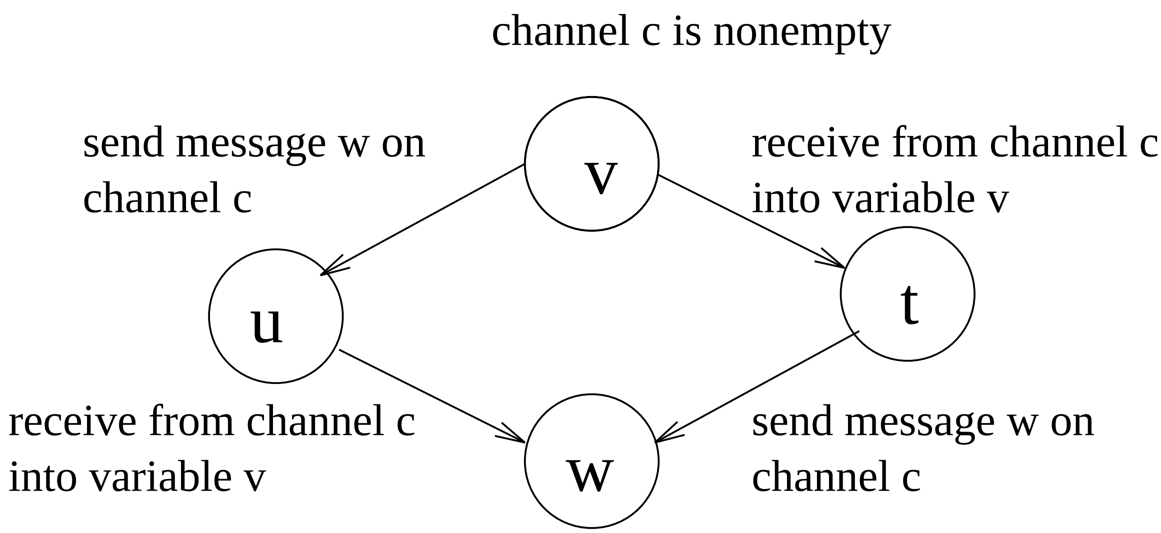

Figure 2: Channels with the Diamond Property

at most one sender token and at most one receiver token) also has the diamond property.

Deterministic Shared Variables Next we describe constructs that allow concurrent processes to share variables so that parallel programs have the diamond property.

Associated with each shared variable is a number of identical tokens. A process can write a shared variable at a point in a computation only if it holds all tokens associated with the variable, at that point. A process can read a shared variable at a point in a computation only if it holds at least one token associated with the variable, at that point. 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.

Deterministic Single-Assignment Variables A useful variant of the deterministic shared variable is the deterministic single-assignment variable (DSAV). A DSAV differs from the deterministic shared variable described in the previous section in that a DSAV is assigned a value at most once in a computation, and execution of a process reading a DSAV is suspended while the DSAV is unassigned.

Associated with each DSAV is at most one writer token. A process can assign a value to a DSAV only if it holds the writer token associated with the DSAV. When a value is assigned to a DSAV, its writer token disappears; thus a DSAV can be assigned a value at most once in a computation.

A pointer to a DSAV can be used to read, but not write, a DSAV. Any process can acquire a pointer to a DSAV. Execution of a read of a DSAV is suspended while the DSAV is unassigned. The writer token associated with a DSAV can be sent from process to process. Likewise, pointers can be sent between processes.

The proof that the DSAV has the diamond property is straightforward.

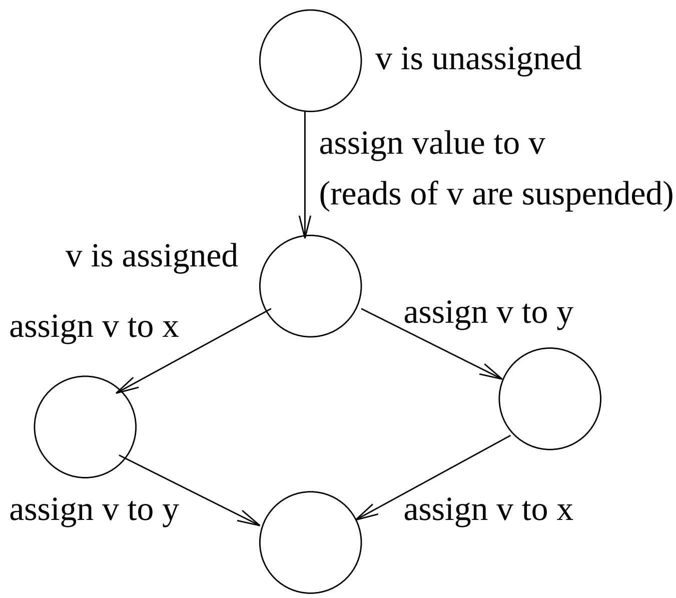

Figure 3: Deterministic Single-Assignment Variables

Though there are other communication protocols that satisfy the diamond property, those described here appear to be adequate for many applications. Next, we describe the communication mechanisms in detail in the context of Fortran M [10]. The ideas described here are, however, language-independent.

3 Processes

The state of a program is defined by a four-tuple: (i) a set P of processes, (ii) a set C of shared variables where a shared variable is a channel, a deterministic shared variable or a deterministic single-assignment variable, (iii) the state of each process in P and (iv) the state of each variable in C.

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 parameters other than tokens of processes 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, and `V`, `W` are the arguments of `P`, and `A`, `B`, `C` are the arguments of `Q`.

The execution of a parallel block in a process `t` causes all the processes in its list of process calls to be created and 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.

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 uninitialised 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 Channels

Types 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`.

For now, assume that processes communicate only by sending and receiving messages on channels, and that processes do not share any other type of variable. (This restriction will be relaxed later.) Therefore, all variables of a process are either local variables of the process or arguments of the process.

Statements that Operate on Channels Next we describe the four additional statements in the extended language dealing with message-passing. The statements for communication 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*.

Example We describe a process `sieve` that is used in a prime number sieve program. The specification of

`sieve(myprime,relatively_prime_in,primes_out)`

is as follows.

1. The value of inport variable `relatively_prime_in` is a receiver token for a channel of type integer. The message sequence received on this channel is the sequence of positive integers, in increasing order, that are less than or equal to `n`, for some arbitrary `n`, and are relatively prime to the first `k` primes, for some `k`, and where `myprime` is the `k+1`-th prime.
2. The sequence of messages sent by the process on the channel with sender token `primes_out` is the sequence of primes that exceed the `k`-th prime and are less than or equal to `n`.

```
    PROCESS sieve(myprime,relatively_prime_in,primes_out)

c   declare parameters of the process
    INTEGER myprime
    INPORT(INTEGER) relatively_prime_in
    OUTPORT(INTEGER) primes_out

c   declare local variables
    INTEGER msg
c   declare ports for internal channels
    OUTPORT(INTEGER) filter_out
    INPORT(INTEGER) filter_in

c   Send myprime on primes_out
    SEND(PORT=primes_out) myprime

c   Discard incoming messages that are divisible by
c   myprime until either the end-of-channel message is
c   received, or a message is indivisible by myprime
    RECEIVE(PORT=relatively_prime_in, END=20) msg
    DO WHILE(divisible(msg,myprime))
      RECEIVE(PORT=relatively_prime_in, END=20) msg
    ENDDO
```

```
c   Since msg is not relatively prime to myprime create a
c   network consisting of processes sieve and filter with an
c   internal channel from filter to sieve. Process filter
c   sends on incoming messages not divisible by myprime.
    CHANNEL(OUT = filter_out, IN = filter_in)
    PROCESSES
      PROCESSCALL filter(myprime,relatively_prime_in,filter_out)
      PROCESSCALL sieve(msg,filter_in,primes_out)
    ENDPROCESSES

    return

c   Close channel corresponding to primes_out and terminate
20  ENDCHANNEL(PORT=primes_out)

    END
c   completes definition of process
```

5 Deterministic Shared Variables

Operations The syntax for declaring 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`. In addition two *inquiry functions* (defined in Fortran90) return attributes of the deterministic shared variable:

1. `TOTAL_TOKENS(y)` is the total number of tokens associated with deterministic shared variable `y`.

2. `TOKENS_HELD(y)` is the number of tokens associated with `y` held by the process in which the function call is made.

An invariant of a process `p` is: `TOKENS_HELD(y)` $= 0$ in `p` if and only if `y` $=$ `NULL` in `p`.

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

Dynamic Storage Allocation of Shared Variables Statements for dynamic storage allocation are similar to allocation statements in Fortran90.

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

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

The first allocate statement is a Fortran90 statement that 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.

Modifying the Number of Tokens The statement

```
SET_TOKEN_COUNT(y,n)
```

where `y` is a deterministic shared variable, and `n` is an integer, can be executed if and only if the process in which the statement is executed holds all the tokens associated with `y`, and `n` is a positive value. A postcondition of this statement is that the number of tokens associated with `y` is `n`.

Deallocation of Deterministic Shared Variables A statement

```
DEALLOCATE(y)
```

where `y` is a deterministic shared variable can be executed at a point in a computation if and only if the process in which the statement appears holds all the tokens associated with `y` at that point; the statement deallocates the space associated with the variable (as in Fortran90).

Sending Tokens Execution of:

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

sends `k` tokens associated with variable `y`. Therefore, if

```
TOKENS_HELD(y) = m
```

before the send, where m $\geq$ k, then after the send:

```
TOKENS_HELD(y) = m - k.
```

An error is posted if m $<$ k before the send because a process cannot send more tokens than it holds.

For convenience, if COUNT does not appear explicitly in the send statement, a default of COUNT = 1 is used; so,

`SEND(PORT = op) y(COUNT=1)` and `SEND(PORT = op) y`

are equivalent.

Receiving Tokens 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.

Moving Tokens A move statement has the form

```
MOVE(y(COUNT = k), z)
```

where y and z are deterministic shared variables in the same process, and k is a positive integer; execution of the statment has the same effect as sending y(COUNT=k) and then receiving that message into z.

Tokens as Process Parameters Tokens are passed to processes as parameters in exactly the same way as sends and receives; the number of tokens held by the caller is decreased and the number held by the called routine is increased by the number specified in the call.

Deterministic Shared Arrays A deterministic shared array is a single object and not an array of deterministic shared elements; therefore, a deterministic shared array has a single set of tokens associated with the entire array. A process that holds a token corresponding to a deterministic shared array can read the entire array, and a process that holds all tokens corresponding to the array can write the entire array.

Implementation on Distributed Memory The deterministic shared variable is an architecture-independent programming construct that can be implemented particularly efficiently on even *weakly coherent* shared-memory architectures. Next, we describe an implementation on distributed memory.

Associated with each shared variable is a master copy. Each token corresponding to the variable contains a pointer to the process and location at which the master copy is stored and the number of tokens associated with the variable.

A process that acquires the right to read, but not modify, a variable is given a read-only copy of the variable. Writes to the read-only copy cause an error to be posted. When a process relinquishes its right to read (and not modify) the variable by sending all its tokens corresponding to the variable, the read-only copy is discarded.

A process that acquires the right to modify a variable is given a read-write copy of the variable. When a process relinquishes the right to modify the variable, by sending away at least one token, its read-write copy is copied into the master copy; then its read-write copy becomes a read-only copy if the process continues to hold at least one token corresponding to the variable, and its read-write copy is discarded if the process no longer holds any token corresponding to the variable.

In a straightforward implementation, the location of the master copy remains unchanged, though in a sophisticated implementation the location can change as computation proceeds.

Copies do not have to be made in a shared-memory system. In a heterogeneous system, such as a network of shared-memory multiprocessors, copies are made if the process that acquires the right to access a shared variable is in an address space that is different from that of the master copy.

Example

Specification Next, we present a very simple example of concurrent processes with shared memory. The example is a 3-dimensional mesh computation. Let n be the dimension of the mesh in some direction, and and let T be the time horizon.

Let x_i^t, for $0 \le i \le n$ and $0 \le t \le T$ be the value of the i-th slice of the mesh at time t (where a slice of a three-dimensional mesh is a two-dimensional mesh). The program computes x_i^t, for $0 < i < n$, and $0 < t \le T$, given the boundary values x_i^0 for all i, and x_0^t and x_n^t for all t, using the formula:

$$x_i^t = f(x_{i-1}^{t-1}, x_i^{t-1}, x_{i+1}^{t-1})$$

The function f is given.

Program The program uses processes indexed i where $0 < i < n$. Each process has two variables that it modifies: `current` and `previous` where on the t-th step of process i, for all i and $t > 0$: `previous` $= x_i^{t-1}$ and the value of `current` computed on the t-th step by process i is x_i^t. Each process has two variables that it reads (but does not modify), and these are `left_value` and `right_value`. On the t-step of process i:

$$\texttt{left\_value} = x_i^{t-1}$$

$$\texttt{right\_value} = x_{i+1}^{t-1}$$

There are 3 tokens associated with `current` and `previous` for each process. At the beginning of step t, process i holds one token corresponding to each of `left_value`, `right_value` and `previous`, and it holds all three tokens of `current`; therefore it can read (but not modify) x_{i-1}^{t-1}, x_i^{t-1}, x_{i+1}^{t-1} and it computes x_i^t.

The code for the steps of process i, other than the boundary processes at either end of the mesh, is:

```
     DO t = 1, T
c        holds 3 tokens of current and 1 token of previous
c        holds 1 token of left_value and 1 token of right_value

c        left_value = x_{i-1}^{t-1}, previous = x_i^{t-1}, right_value = x_{i+1}^{t-1}
         CALL F(left_value, previous, right_value,current)
c        current = f(left_value, previous, right_value)
c        current = x_i^t

c        send one token each of left_value and current to left
         SEND(PORT = to_left) left_value, current

c        send one token each of right_value and current to right
         SEND(PORT = to_right) right_value, current

c        receive one token each of left_value and previous from left
         RECEIVE(PORT = from_left) previous, left_value
c        left_value = x_{i-1}^t

c        receive one token each of right_value and previous from right
```

```
      RECEIVE(PORT = from_right) previous, right_value
c     right_value = x^t_{i+1}

c     process holds one token each of left_value, right_value
c     process holds one token of current and 3 of previous

      CALL interchange(current, previous)
   ENDDO
```

6 Deterministic Single-Assignment Variables

We describe deterministic single-assignment variables (DSAV) in Fortran M briefly. Associated with each DSAV is at most one writer token and an arbitrary number of reader tokens. We use reader tokens rather than pointers because pointers in Fortran 90 can be used to read and write, and we want to emphasise that a reader token provides the limited capability of reading but not writing. A DSAV is created (i.e., allocated in Fortran 90 terms) with a single writer token and a single reader token. Reader tokens can be duplicated whereas writer tokens cannot be duplicated. Assigning a value to a DSAV destroys the writer token associated with it.

The syntax for declaring writer and reader tokens and for allocating DSAVs is as follows:

```
c     declare variables
      REAL, WRITER :: w
      REAL, READER :: r

c     allocate variables
      ALLOCATE((WRITER = w, READER = r))
```

Execution of a statement that reads `r` such as `x = r+5` is suspended while the DSAV corresponding to `r` is unassigned, and when the DSAV becomes assigned the statement is executed. Reader and writer tokens can be passed along channels in the usual way. A reader token `r0` is duplicated to obtain a duplicate `r1` as follows:

```
      DUPLICATE(r0,r1)
```

The proof that these constructs are deterministic is straightforward.

We do not give examples of single-assignment programs. There are several languages that use single-assignment and execute on distributed-memory machines such as Sisal [4], Strand [11], and PCN [5].

7 Nondeterminism

A programmer may want to allow potential nondeterminism to improve efficiency or to make programs simpler. For instance, a programmer may want to design a process `p` to accept messages from either process `q` or process `r`, in arbitrary order. Fortran M has two nondeterministic constructs: `MERGER` and `PROBE` for such situations. In addition, Fortran M uses the `INTENT(IN)`, `INTENT(OUT)` mechanism of Fortran 90 to pass parameters to processes; this parameter-passing mechanism is potentially nondeterministic because the same variable may be read and modified by concurrent processes.

A method for designing parallel scientific applications is to first develop and debug the application using a language with compiler support for verification of determinism, and later add nondeterministic constructs (if necessary) to improve efficiency.

MERGER A `MERGER` is similar to a channel except that it can have more arbitrary (positive) number of output ports; like a channel, a `MERGER` has a single input port. A statement:

```
MERGER(OUT = op1, OUT = op2, IN = ip)
```

creates a merger linking output ports `op1` and `op2` with input port `ip`. The sequence of messages delivered to input port `ip` is a fair merging of the sequences of messages sent on output ports `op1` and `op2`.

PROBE The command:

```
PROBE(PORT = ip, EMPTY = v)
```

where `ip` is an input port, and `v` is a boolean variable, sets `v` to `false` only if there is a message in the channel corresponding to input port `ip` [17]. Variable `v` can be set to `true` by the command, even if there is a message in the channel (because the message is still in transit and has not yet arrived at the input port `ip`). All messages sent arrive at the input port eventually, and therefore, the following loop will terminate if there is a message in the channel associated with input port `ip`:

```
v = .TRUE.
WHILE(v) DO
   ....
   PROBE(PORT = ip, EMPTY = v)
ENDDO
```

8 Earlier Work

The contribution of this paper is to incorporate well-known ideas about the Church-Rosser theorem, capabilities, channels, distributed shared memory and single-assignment variables into a widely-used sequential language to get 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 [22, 12, 1] and high-level languages [4] highlights some of the weaknesses and strengths of Fortran M. 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 [22, 12] and applicative languages [4] do not require the programmer to deal with processes, messages or tokens.

Applications such as multidisciplinary design require task-parallel coupling of data-parallel programs. Fortran M can be used to provide such coupling. Data-parallel communications on arrays of channels provide a simple mechanism for coupling multiple data-parallel programs.

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 [3] or PVM [21] 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.

Debugging Fortran M programs is simpler than debugging parallel programs that use message libraries because replaying Fortran M programs requires only that nondeterministic choices made at the `MERGER` and `PROBE` constructs be recorded in a computation to obtain a replay of that computation. Replay of programs that do not use `MERGER` or `PROBE` require nothing special, because such programs are deterministic. Subtle race conditions can occur in processes communicating using message libraries, and replay requires recording every resolution of a potential race condition in a computation.

A major difference between Fortran M and actor-based languages [19, 2] is the Fortran M focus on separating deterministic and nondeterministic constructs.

An implementation of Fortran M (with channels but without shared variables) is available from anonymous ftp server `info.mcs.anl.gov` (directory `/pub/fortran-m`) at Argonne National Laboratories.

Fortran M has been used to develop libraries of parallel programs in linear algebra, spectral methods, mesh computations, computational chemistry, computational biology, and to explore the integration of task-parallel and data-parallel programming.

References

[1] Bagrodia,R., and S.Mathur, "Efficient Implementation of High-Level Parallel Programs," *Proc. ASPLOS-IV*, April, 1991.

[2] Bagrodia,R. and W.Liao, "Maisie User Manual," Tech. Report, Computer Science, UCLA, Los Angeles, Calif. 1990.

[3] 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.

[4] 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, pp 612-616.

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

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

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

[8] Dennis, J.B., and E.C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Comm. ACM*,9,Mar. 1993, pp 143-155.

[9] Feo,J.T. ed., *A Comparitive Study of Parallel Programming Languages: The Salishan Problems*, Special Topics in Supercomputing, Vol.6, Elsevier Science Publishers B.V., The Netherlands.

[10] Foster, I. and K. M. Chandy, "Fortran M: A Language for Modular Parallel Programming," Preprint MCS-P237-0992, Argonne National Laboratory, Argonne Ill. 60439, 1992.

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

[12] 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.

[13] Hoare, C. A. R., Communicating Sequential Processes, *CACM*, 21(8),1969, pp 666-677.

[14] Jones, A.K., "Protection in Programmed Systems," PhD. Thesis, Computer Science, Carnegie-Mellon University, 1973.

[15] Lampson,B.W., "Protection," *Proc. 5th Annual Princeton Conf. on Information Science Systems*, 1971, pp 437-443.

[16] Li,K., and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Comp. Systems*,7(4),Nov.1989, pp 321-359.

[17] Martin, A. J., "The Probe: An Addition to Communication Primitives," *Information Processing Letters*,20, April 1985, pp125-130.

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

[19] Seitz,C.L., "Multicomputers," in *Developments in Concurrency and Communication*, ed. C.A.R. Hoare, Addison-Wesley, Reading, Mass., 1991.

[20] Silberschatz,A., J.Peterson and P.Galvin, *Operating Systems Concepts*, Addison-Wesley, Reading, Mass, 1991.

[21] Sunderam,V., "PVM: A Framework for Parallel Distributed Computing," *Concurrency Practice and Experience*,2,1990, pp 315-339.

[22] Thinking Machines Corporation, *CM Fortran Reference Manual*, Thinking Machines, Cambridge, Mass., 1989.