ON VARIABLES AS ACCESS SEQUENCES IN PARALLEL ASYNCHRONOUS COMPUTATIONS

Craig Williams
Paul F. Reynolds, Jr.

Computer Science Report No. TR-89-17
December, 1989

On Variables as Access Sequences In Parallel Asynchronous Computations*

Craig Williams
Paul F. Reynolds, Jr.

ABSTRACT

We introduce a new method for coordinating access to shared variables in parallel asynchronous computations. The method is based on access sequences, the representation of each shared variable as the sequence of values written to and read from the variable, and on parallel operations, a mechanism for accessing groups of shared variables atomically. Parallel operations on access sequences replace locks as the basis for implementing atomic actions and process synchronization. Advantages include reduction of the potential for deadlock and starvation and greater concurrency in accessing shared memory. The principal costs are the increase in space needed to represent shared variables and in the complexity of the interconnection network and the memory modules.

1. INTRODUCTION

The essence of a variable is that it changes, that it exists in time. We propose that it can be useful to think about a variable as the sequence of values written to and read from the variable during a computation and, in an asynchronous parallel computation, to represent shared variables as access sequences. Representation of variables as access sequences, together with parallel operations, a mechanism for accessing groups of shared variables atomically without locks, form the basis for a simple and efficient way for asynchronously executing processes to coordinate their accesses to shared memory. Parallel operations are defined in a companion paper [RWW89].

Asynchronous processes need two types of coordination in accessing shared memory. First, processes must be able to access groups of shared variables as an atomic action, without interference from other processes. (We do not use the term "mutual exclusion" only because it implies a mechanism we do not use.) Second, processes must be able to synchronize to ensure a specific ordering among accesses to shared data. An example of a synchronization pattern is the producer/consumer relationship.

* The work of Craig Williams was supported by an assistantship in parallel processing sponsored by DARPA and NASA and administered by the University of Maryland.

Parallel operations on access sequences provide a mechanism for handling both types of coordination without the use of locks. Concurrency is restricted, but by data dependencies, not locks. Eliminating locking as the basis for coordinating access to shared variables brings many benefits, including greater concurrency in accessing shared variables and elimination of the potential for deadlock implicit in locking. The principal costs are the increase in the complexity of the memory modules implied by the access sequence representation, and in the space needed to represent variables. In section 6, we describe a practical implementation of access sequences and show that the entire access sequence need not actually be stored.

Our proposal for coordinating accesses to shared memory is based on three ideas:

(1) Variables can be represented as sequences of values instead of as single values. Each element in the sequence represents an access to the variable. An element can contain a value read from or written to the variable or can reserve a position for such a value.

(2) The execution of an access to a shared variable can be split into a scheduling step and an assignment step. The scheduling step reserves a position in the variable's access sequence and the assignment step transfers a value to, in the case of a write, or from, in the case of a read, the position reserved by the scheduling step.

(3) A process can schedule accesses to multiple shared variables as an atomic action using parallel operations.

A process executes an atomic action by issuing a parop scheduling all the shared variable accesses required by the atomic action, executing the assignment steps for these accesses as the values become available. The use of the parallel operation in the scheduling step reserves a consistent ''slice'' across the sequences of the shared variables accessed in the atomic action. A process synchronizes with another process by scheduling a write (read) to a shared variable immediately succeeded by a read (write) where either the write or read is executed by the other process.

There is an important limitation on the use of parallel operations in scheduling atomic actions: a process must be able to name the shared variables the atomic action will access before it executes the atomic action. If a process must both read and use a pointer or index in the same atomic action it will have insufficient information to schedule the atomic actions. We propose three methods of accommodating these troublesome atomic actions. One of these methods involves introducing a form of locking we

call RW-locking. We are able to prove that atomic actions that do not use RW-locking are deadlock-free. Programs with atomic actions that are implemented using RW-locking must be independently proven deadlock-free.

Our focus is on asynchronous, shared memory computations. For concreteness, we assume a topology in which a bank of processing elements (PEs) is connected by an Omega interconnection network to a bank of memory modules (MMs). The use of parallel operations imposes the requirement that all processes access shared memory through the same network. We discuss extending this method of coordinating parallel computations to synchronous computations in section 5 and to other topologies and message-based computations in section 8.

In the next section, we describe each of the three ideas that form the basis for our proposal for coordinating access to shared memory. Section 3 defines the operations on access sequences. In sections 4 and 5, we describe how these operations can be used to implement atomic actions and synchronization. Section 6 proposes an implementation of variables as access sequences. In the remaining sections, we review related work and discuss topics for future research.

2. THE BASES FOR ATOMIC ACTIONS

The following three ideas form the basis for our proposal for coordinating access to shared variables:

(1) Variables can be represented as sequences of values instead of as single values. Imagine that each shared variable is represented by a stack of children's alphabet blocks, each with a value, the same value for a given block, stamped on each face. An assignment to the variable adds a new block to the top of the variable's stack. Representing variables as single values can be compared to looking at the stacks only from directly overhead. In this one dimensional view, only the top block in each stack is visible. Representing variables as sequences corresponds to looking at the stacks from the side. All the blocks in every stack are visible in this two dimensional view.

An immediate advantage of the access sequence representation is the additional freedom this extra dimension gives processes in accessing variables. In a parallel asynchronous computation, the order in which processes access a given shared variable is constrained by the order in which the processes accessed other shared variables. As an example, consider the following atomic actions executed concurrently by processes P_1 and P_2:

```
P1::  IF V1 THEN read(V2);
P2::  V1 := FALSE; V2 := 10;
```

Assume initially V1 = TRUE and V2 <> 10. If P_1 accesses V1 before P_2, then P_1 must read V2 before P_2 writes V2, i.e., P_1's access to V2 must logically precede P_2's access. If variables are represented as single values, then P_1's access to V2 must not only logically precede P_2's access, but must also actually precede P_2's access. Otherwise P_2 will overwrite the value of V2 that P_1 must read. The access sequence representation allows the accesses to occur in either order. The "old" value of V2 is still visible even after P_2 completes its assignment. Thus the access sequence representation allows accesses to be executed "out of sequence", in an order that is different from the order that would otherwise be required for correct execution of atomic actions. In particular, writes can occur as soon as the process executing the write computes the value to be written. The write will not obscure a value that must be seen by subsequent reads that logically precede the write. This idea of maintaining old versions of data to accommodate tardy reads has been well studied and is the basis for multiversion concurrency control in databases. (See section 7.)

An additional advantage of the access sequence representation is that it provides the structure for scheduling accesses as atomic actions. A access sequence not only records values for each write made to the variable, but also records reservations for writes and reads. As we will explain, these reservations allow tardy writes to be accommodated as well as tardy reads.

(2) The execution of an access to a shared variable can be divided into two parts: scheduling and assignment. Scheduling reserves the context for a read or write by reserving a position in the access sequence. Note that the scheduling step for a write can be executed before the value to be written has

been determined. Assignment transfers a value. For writes, the transfer is from a local variable or register to the position reserved for the write in the access sequence. For reads, the direction of the transfer is reversed. The assignment step for writes is initiated by the MM containing the variable when the value of the variable at the position reserved by the read has been determined. This value is determined when the assignment step is executed for the write preceding the read in the access sequence.

Note that the assignment steps for scheduled accesses can occur at any time subject to one constraint — the assignment step for a scheduled read must occur after the assignment step for the highest write under it in the stack. This freedom in the order in which assignment steps can be executed is one source of additional concurrency over that allowed by methods of coordinating access to shared memory that rely on locking.

The reason for dividing accesses into a scheduling step and an assignment step is that the separation allows processes to use parallel operations to schedule atomic actions. If scheduling could not be separated from assignment, scheduling would have to be piecemeal. It would be impossible to guarantee that a group of accesses could complete without either rollback or interference owing to intervening accesses by other processes.

Another reason for dividing accesses into a scheduling step and an assignment step is synchronization. Not only can scheduling and assignment occur at different times, they can be performed by different processes. Using operations on access sequences, a process can schedule a write (read) and an immediately succeeding read (write) where the assignment for either the read or write is to be executed by another process.

(3) A process can operate on multiple shared variables as an atomic action using parallel operations. Parops are described in a companion paper [RWW89]. A parallel operation (parop) is a special case of the atomic action, a set of one or more operations on shared variables executed as an indivisible step. The parop is not as general as the atomic action because each operation must be independently issuable, i.e., the process issuing a parop must be able to issue all the operations in the parop before any operation in

that parop is executed. In this paper, we assume that parops are composed of operations that schedule or execute assignments on access sequences. The operations on access sequences are defined in the next section.

Every execution of a parop program is ''sequentially consistent''. A computation is sequentially consistent if it is equivalent to an execution in which the atomic actions are executed serially in an order that is consistent with the order specified by each process's sequential program [ShS88]. Parops can be implemented without locking and can be pipelined.

We can now explain informally how these ideas combine to support atomic actions. A process executes an atomic action by issuing a parop scheduling all the accesses to shared variables required for the atomic action, executing the assignment steps for these accesses as the values become available. For example, a process, P_1, executes the atomic action

```
P1:: V1 := V1 * V2                    V1 and V2 are shared variables
```

by first issuing a parop naming `V2` for a read access and `V1` for a read access followed by a write access. Note that execution of the assignment steps is controlled by data flow. For a read this means waiting until the immediately preceding write scheduled to access the variable completes its assignment and the MM returns the value assigned. For a write, the assignment can be executed as soon as the value to be computed becomes available. Typically this computation will require the completion of reads in the same atomic action. In the example, when P_1 receives back the values for `V1` and `V2` it can compute the new value of `V1` and execute the assignment step for the write it has already scheduled to `V1`.

Execution of an atomic action scheduled with parops is atomic because the parop that schedules the atomic action reserves a position in the access sequence of each of the accessed shared variables that is consistent with indivisible execution of the atomic action. Even though the accesses to shared variables in an atomic action may occur at widely separated times and places, they appear to every process in the computation to be executed as an indivisible operation.

3. OPERATIONS ON ACCESS SEQUENCES

The access sequence of a shared variable v is the sequence of values written to and read from v during a computation. An access sequence is similar to an access trace or log, but is in the order in which the accesses are scheduled, rather than the order in which assignments are executed. Initially, v's access sequence is empty. The scheduling step for a read or write to v appends an element to v with the special value TBA "to be assigned". In the case of a read, the pId of the process to which the value of v is to be returned is also recorded as a component of the value of the element appended by the read. The assignment step for a write changes the value from TBA to the value assigned. We say that a write (read) is "unsubstantiated" if the scheduling step has been executed but not the assignment step and that a write (read) has been "substantiated" when the assignment step has been executed. The assignment step for a read changes the value from TBA(pId) to the value assigned to v by the "preceding" write when that write is substantiated. The write "preceding" a given read to v is the last write that appended an element to v before the element appended by the read. We say that a read "succeeds" a given write if the write is the preceding write for the read. A read to an empty sequence is undefined. Note that reading an empty sequence corresponds to reading an uninitialized variable.

The scheduling and assignment steps can sometimes be collapsed. For a write accessing v, it is convenient to define an operation that appends an element to v with the value assigned to v in a single step. (This operation is the w operation defined below.) For a read, if the preceding write is already substantiated when the read is scheduled, the assignment step can be executed immediately. In this case, a practical implementation will skip the scheduling step that would otherwise append an element with the value TBA(pId) and instead immediate return to process pId the value assigned by the preceding write.

The operations we define are on variables, where it is understood that a variable is of a primative data type, not a data structure, such as a record or an array. As a simplifying assumption, we assume that each shared variable is located in a single MM and that the hardware supports indivisible reads and writes to individual variables. The assumption can be relaxed by representing each shared variable in the pro-

gram with one or more "machine-unit" shared variables, each of the size for which the hardware supports indivisible access, and treating each access to a shared variable in the program as an atomic action that accesses each of the "machine-unit" variables constituting that variable.

We define five operations on access sequences. The operations are intended to be executed by the memory modules (MMs) on access sequences implemented as arrays. We assume initially that the arrays are infinite. Section 6 describes a practical implementation. The five operations are as follows:

SCHED(V,pos,pID)
: Schedules a write. Appends an element to `V`'s access sequence and sends to process `pId` (which may be different from the process that issued the operation) the address of `V` and the position, pos, of the newly appended element in `V`'s access sequence. Process `pId` will use these values as parameters to the `ASSIGN` operation when it executes the assignment step for the write.

ASSIGN(V,pos,val)
: Executes the assignment step for a previously scheduled write. Assigns the value `val` to element `pos` of `V`'s access sequence and propagates the value to the succeeding reads.

R(V,pos,var,pID)
: Executes a read access, assigning the local variable `var` the value of `V`. If the preceding write is already substantiated, the `R` operation immediately sends to process `pId` the address of `V` and the value assigned by the preceding write. Otherwise, the `R` operation appends an element with the special value `TBA(pId)` and sends to process `pId` the position, pos, of the new element in the access sequence. The value assigned to `V` by the preceding write will be sent to process `pId` as a result of a later `ASSIGN` or `CANCEL` operation.

W(V,val)
: Schedules and executes the assignment for a write in a single step. The `W` operation appends an element to `V`'s access sequence with the value `val`.

CANCEL(V,pos)
: Cancels a previously scheduled write. The `CANCEL` operation changes the value of the `pos`th element in `V`'s access sequence from `TBA` to `DELETED`. If the preceding write is substantiated, the `CANCEL` operation propagates the value of that write to succeeding reads. When an `ASSIGN` operation propagates, the value propagates through elements marked `DELETED`.

To enable the process receiving a value returned from a `R` operation to associate the value with the correct local variable, the MM tags values it returns from an `R` operations with both the variable name and the position of the read in the access sequence. The access sequence position must be included in the tag because a process may have more than one unsubstantiated read scheduled for the same variable. A table for each process is maintained locally (by the process, the operating system, or the processor network interface), recording for each unsubstantiated read the name of the shared variable, the local variable that is to receive the value, and, when the MM supplies the value, the position of the read in the

variable's value sequence. Since the process itself does not typically refer to the value of the `pos` parameter of the `R` operation, we will usually substitute a hyphen for this parameter.

The `CANCEL` operation cancels only write accesses. We do not propose an operation on access sequences to cancel reads because the cost of cancellation is comparable to the cost of executing the read and because the cancellation may be ineffective — unlike a write, a read may be substantiated before the cancel arrives at the MM. The /*vdiscard/*w operation, an operation that cancels reads, but that is executed locally at the PE is introduced in a later section.

Note that if the third parameter in a `SCHED` or `R` operation specifies a pID that is different from that of the process executing the operation, then the assignment step will be executed by another process. We call such operations "synch operations". For example, the operation `R(V,-,val,2)` is a synch operation if it is issued by a process other than P_2. The operation sends the value of the current element of `V` to P_2 even if the operation is executed by another process. We treat the `pId` parameter of the `SCHED` and `R` operations as a default parameter. When omitted it is assumed to specify the pID of the process that executes the operation.

We do not intend that these operations be used directly by the programmer. Ideally, the programmer will use higher level constructs that will be translated automatically into access sequence operations. We define notations for specifying atomic actions and send/receive synchronization in the following two sections, but our primary purpose is not to define a programming language but to show that parallel operations on access sequences can be used as a basis for coordinating access to shared memory.

This is a basic set of operations. In section 8 we briefly describe other operations that we intend to consider adding to the operation set.

4. ATOMIC ACTIONS

An atomic action is a group of statements executed by a single process as an indivisible unit from the point of view of any other process in the computation [OwL82]. Within an atomic action, the computation can be analyzed as if it were a sequential computation. No other process can interfere with an

atomic action or observe intermediate states of the execution of the atomic action.

An atomic action may not contain a synch operation. Communication with another process from within an atomic action violates the definition of an atomic action.

The simple way to implement atomic actions in a shared memory computation is to use locks. Before executing an atomic action, a process sets locks that prevent other processes from accessing the shared variables named in the atomic action, releasing the locks upon completion of the atomic action. Locks can be placed on the shared variables directly or on a critical section, the code that references the shared variables. The monitor typifies this latter approach to locking. Locking limits concurrency beyond the limitation implied by data dependencies and creates the potential for deadlock and starvation.

Parallel operations on access sequences give an alternative way of implementing atomic actions. Instead of locking out other access to variables accessed in an atomic action, the process reserves a slice of each variable's access sequence, using parops to ensure that the slice is consistent with indivisible execution of the atomic action. Advantages of this approach are greater concurrency and freedom from deadlock and starvation.

In this section we define a notation for designating an atomic action and describe how atomic actions are implemented using parallel operations on access sequences.

4.1. Notation

Atomic execution of a group of statements is specified by inclosing the statements within curly braces:

```
{ an atomic action }.
```

Atomic actions may not contain other atomic actions — nesting of curly braces is undefined.

Since we assume that the hardware supports indivisible reads and writes to individual variables (scalars), a single reference to an individual shared variable is the trivial case of the atomic action. We omit curly braces in the trivial case. For example, to indicate that the value of `v2` is to be read as an

atomic action and then, in a subsequent atomic action, written to `V1`, we write

```
V1 := V2
```

instead of

```
{var := V2};  {V1 := var}.                    var is a local variable
```

Note that there are no curly braces around the the first statement, `V1 := V2`. The statement specifies that the read to `V2` and the write to `V1` are each executed as an atomic action, but not as part of the same atomic action.

4.2. Translation of atomic actions

We define the "write set" of an execution of an atomic action to be the set of shared variables written by the atomic action. The "read set" is defined analogously, except that a variable is not in the read set if the first read in the atomic action to the variable occurs after the first write. These access sets are defined with reference to a given execution because different executions of the same atomic action may have different access sets. The name of a variable or the type of access to be made to the variable may depend on the value of another variable accessed in the same atomic action. Later in the section, we describe methods for translating atomic actions with access sets that can only be determined dynamically. For now we assume that the access set of an atomic action can be statically determined.

An atomic action can be translated into a sequence of one or more parops, interspersed with statements specifying local computation, where each parop is composed of one or more operations on an access sequence. The first parop in the sequence, the "scheduling parop", contains a `SCHED` operation for every variable in the write set and an `R` operation for every variable in the read set. The scheduling parop may also include `W` operations if the process has already computed the value to be assigned. Subsequent parops contain `ASSIGN` operations for `SCHED` operations contained in the scheduling parop.

The scheduling parop need contain at most two operations for any given shared variable even if the atomic action accesses the variable more than twice. Only the first read and the last (as determined by the

sequential flow of control within the atomic action) write to any one variable need be scheduled (and no read need be scheduled if the first read occurs after the first write). Since no other process can interfere with an atomic action, the other accesses can be to a local copy of the variable. If the atomic action first reads a variable V and then writes V, the scheduling parop will contain the operations

```
< R(V,pos,var) || SCHED(V,pos) >.
```

The angle brackets signify that the order in which the operations are listed in the parop is the order in which the operations will be executed at the MM. Operations inclosed by a pair of angle brackets must all name the same shared variable. In the absence of angle brackets, operations within a parop may be executed in an arbitrary order.

4.3. Execution of atomic actions

A process begins execution of an atomic action by issuing the scheduling parop. As the values become available from which to compute the values to be assigned, the process issues an ASSIGN operation for each SCHED operation, completing each write access. An atomic action is ''completed'' when the assignment step for every access scheduled by the scheduling parop has been executed. Unless the process has insufficient information to schedule the next atomic action until the current atomic action is completed, a process is not required to complete an atomic action before issuing the scheduling parop for the next atomic action. In an efficient implementation, each process will overlap the assignment phase of the current atomic action with the scheduling of the next.

Whether an R operation is blocking is an implementation choice. Blocking is the simplest alternative. A process blocks until all reads issued in the scheduling parop are satisfied, i.e., until the process receives back from the MM values for each R operation issued in the scheduling parop. This choice unnecessarily limits concurrency and prevents pipelining atomic actions. A better choice is to allow a process to proceed until it must use a value not yet returned by an R operation. If this latter alternative is chosen, care must be taken that the process continues to ''listen for'' the return of previously issued R operations. This requirement is implied by the write-substantiation rule.

4.4. The write-substantiation rule

Once memory has returned values for every R operation issued by a process in a parop, PO, and in parops issued by the process before PO, the process must be able to issue an ASSIGN operation for every SCHED operation it issued in PO. We refer to this requirement as the "write-substantiation" rule. The write-substantiation rule is important to ensure that execution of atomic actions is correct and free of the potential for deadlock and starvation.

The rule imposes three restrictions — one on the implementation, the other two on the program, as follows:

(1) The process must "listen for" the return of values from unsubstantiated R operations. In particular, if the process executing an atomic action is blocked, it must be reactivated when the last unsubstantiated R operation of its oldest uncompleted parop returns a value. (This is the minimal requirement, not a recommended implementation. A rule that is likely to be easier to implement and that conforms with this restriction is that a process is reactivated whenever a value is returned to it by an R operation.)

(2) For every SCHED operation contained in parop PO, the value to be assigned in each corresponding ASSIGN operation (or the information needed to cancel the write reservation) must depend only on values known to the process that issues PO when every R operation in PO and in parops issued by the process before PO is substantiated. In particular, values assigned in write accesses scheduled in PO must not depend on values returned by R operations contained in parops issued after PO.

(3) Every write access scheduled by an atomic action must be substantiated by the atomic action. In terms of operations on access sequences, for every SCHED operation issued in an atomic action, the process must issue a corresponding ASSIGN or CANCEL operation.

It is possible to write programs that violate the last two restrictions by directly specifying operations on access sequences (the next subsection contains an example), but is difficult or impossible in a typical procedural programming language to violate either of these restrictions. If all three restrictions are observed,

execution of an atomic action conforms to the write-substantiation rule.

4.5. Correctness

Concurrent execution of a group of atomic actions is "correct" if the execution is equivalent from the point of view of every other process in the computation to a serial execution of the same atomic actions in an arbitrary order. (Of course, if the program specifies that atomic action A must precede atomic action B, then the execution must be equivalent to some serial execution in which A precedes B.) The serializability of the execution of atomic actions implies that the execution of each atomic action is indivisible from the point of view of any other process in the computation.

Concurrent execution of a group of atomic actions using parallel operations on access sequences is serializable if each atomic action conforms with the write-substantiation rule and begins execution by issuing a scheduling parop. The sequential consistency of parops ensures that the execution of a set of parops induces a partial order over the parops. If every atomic action in a computation conforms to the write-substantiation rule, concurrent execution of a group of atomic actions is equivalent to a serial execution of the same atomic actions in an order that extends the partial order induced by the execution of their scheduling parops.

Correctness depends on conformity with the third restriction imposed by the write-substantiation rule — for every SCHED operation issued by an atomic action, the atomic action must issue a corresponding CANCEL or ASSIGN operation. Violation of this restriction can cause a serial execution of a group of atomic actions to block even though a concurrent execution of the same atomic actions terminates. As an example of the need for conformity with the write-substantiation rule, consider the atomic actions A, B, C, and D below. Atomic actions A and B executed by process P_1 assign the value of V2 to V1. Note that A violates the write-substantiation rule by containing a SCHED operation with no corresponding ASSIGN operation.

```
P1:: { SCHED(V1,pos) };                          atomic action A
     { R(V2,-,var); ASSIGN(V1,pos,var) };        atomic action B
P2:: { R(V1,-,var1) || R(V2,-,var2) }            atomic action C
P3:: { W(V2,5) };                                atomic action D
```

Assume that the partial order (in this case also a total order) induced by the execution of the scheduling parops for these atomic actions is ACDB. Execution of these atomic actions using parallel operations is not equivalent to serial execution in this (or any other) order. The concurrent execution terminates but a serial execution in the order defined by the execution of the scheduling parops blocks. The preceding write for the R operation on V1 in C can not be substantiated until B is executed, but B can not be executed in a serial computation until C completes. In effect, P_1 creates a twist in time, reaching into P_2's future to bring back a value (the value of V2) that it inserts into P_2's past (as the value of V1). In the reduced computational space of a serial computation, there is not enough room for this twist and the execution blocks. Conformity with the write-substantiation rule implies that no such twist can occur — that each atomic action will be able to complete when executed serially in an order that is consistent with the partial order induced by execution of their scheduling parops.

4.6. Deadlock-freedom

Atomic actions that conform with the write-substantiation rule are deadlock-free. Deadlock implies a cycle in the ordering among parops and the sequential consistency of parops implies that such a cycle can not exist.

If the computation is deadlocked, then there is a cycle of processes such that each process is waiting on the process that precedes it in the cycle. Assuming processes P_i and P_j are both executing atomic actions using parallel operations on values sequences, P_i waits on process P_j if and only if P_i is waiting for a value to be returned by an R operation, where the preceding write for the R operation was scheduled by a SCHED operation issued by P_j. Each blocked process in the cycle is waiting on an R operation that was issued in the scheduling parop for the process's oldest uncompleted atomic action. Otherwise, by the write-substantiation rule, the oldest uncompleted atomic action could be completed. Therefore, a cycle of blocked processes implies a cycle in the execution order of the set of parops containing of the scheduling parop for the oldest uncompleted atomic action for each of the blocked processes. But the sequential consistency of parops ensures that the order in which parops are executed is acyclic.

4.7. Liveness

Assuming that every write is eventually substantiated or cancelled, atomic actions are starvation-free — no process that attempts to execute an atomic action will be blocked forever. All operations on a shared variable are executed by the MM in FIFO order by arrival time at the MM. Thus every atomic action will be scheduled in the order in which the scheduling operations arrive at the MM. The assignment step for each read access will eventually be executed because the condition for the read — substantiation of the preceding write — will eventually be true. Our assumption that every write is eventually substantiated is valid unless the code specifies a SCHED operation with no corresponding ASSIGN or CANCEL operation or a process deadlocks, fails, or enters a nonterminating loop before substantiating or cancelling a scheduled write.

Note that starvation in the form of livelock can occur in implementations of atomic actions in which atomic actions may rollback [Ree83] or in implementations that rely on the test-and-set primitive to acquire locks or to ensure mutually exclusive execution of P and V operations.

4.8. Dynamic access sets

The method we have described for implementing atomic actions requires that the read and write sets of the atomic action be known before execution of the atomic action begins. As indicated above, this information may not be available. The access sets may not be known at the beginning of the atomic action because the name of a variable to be accessed or the type of access to be made may depend on the value of another variable in the same atomic action. We call a variable whose value determines the access set of an atomic action an "access variable". Access variables either appear in boolean expressions that determine the flow of control or are used in address computations, e.g., pointers and array indices.

We propose three techniques for executing atomic actions with access sets that can only be determined dynamically.

THE CANCEL TECHNIQUE. In some atomic actions, the process can schedule accesses to every variable

that *may* be accessed, using the CANCEL operation to cancel write reservations that are later determined to be unnecessary. Consider the atomic action

```
{ IF V1
    THEN V2 := 5
    ELSE V3 := V4 }
```

Until the access variable V1 is read, the access sets are unknown. (If V1 is true, the write set contains V2 and the read set contains only V1, otherwise the write set contains V3 and the read set contains both V1 and V4.) This type of atomic action can be executed with parallel operations on access sequences by issuing a scheduling parop based on the assumption that all branches of the condition will be executed, canceling unneeded SCHED operations after the condition is evaluated. In the example above, the process can reserve space in all of V2, V3, and V4's access sequences, later cancelling the unused write reservation:

```
R(V1,-,val1) || SCHED(V2,v2pos) || SCHED(V3,v3pos) || R(V4,v4pos,val4)
IF val1
   THEN ASSIGN(V2,v2pos,5) || CANCEL(V3,v3pos) || discard(V4,v4pos)
   ELSE ASSIGN(V3,v3pos,val4) || CANCEL(V2,v2pos)
```

A `discard(V,pos)` operation is a locally executed operation that cancels a read access. It updates the process's locally maintained table of unsubstantiated reads to indicate that a value sent to the process by main memory tagged `(V,pos)` should be discarded. After executing a `discard(V,pos)` operation the process is free to issue another R operation naming the same local variable to receive the value. The local variable will be assigned the value returned by the second R operation even if it arrives at the process network interface before the value returned by the first R operation. The discard operation was not necessary in this example, but can be necessary in the execution of atomic actions that contain loops.

The cancel technique is designed for atomic actions in which control flow creates the uncertainty over the access sets, as in the example above, and can also be appropriate for atomic actions that read a variable and use it to index into a small array. Performance limits the use of the cancel technique to cases in which the process can identify in advance a superset of each of the actual access sets that is "not much larger" than the access set. If every process routinely reserves all of shared memory for each atomic

action using this technique, the computation would be serialized. At some point, as the size of this superset grows in relation to the size of the actual access set, other techniques are needed.

THE REPEATED READ TECHNIQUE. Atomic actions that contain pointers or indices into large arrays are more troublesome because there is no way to identify a "not much larger" superset of an access set. An example of an atomic action in which the access sets are unknown because a pointer is read and then dereferenced in the same atomic action is as follows:

```
{n := c^.next; n^.data := e}
```

Until the access variable `c^.next` is read, the process has insufficient information to schedule the write access. Atomic actions that contain such pointers and indices can be executed by using repeated reading. The process executing the atomic action first reads the access variable (or variables) and then issues the scheduling parop for the atomic action based on the value read. The scheduling parop must contain a second read of the access variable to confirm that the value has not changed. If it has, the process must cancel every write and discard every read scheduled by the scheduling parop and try again.

The atomic action above can be executed using the repeated read technique as follows:

```
R(c^.next,-,n);                                          the first read
done := FALSE;
REPEAT
    R(c^.next,-,n') || SCHED(n^.data,pos);               the scheduling parop contains the confirming read
    IF n = n'
       THEN BEGIN
            done := TRUE;
            ASSIGN(n^.data,pos,e)
            END
       ELSE CANCEL(n^.data,pos) || R(c^.next,-,n);       the confirming read for the next iteration
UNTIL done;
```

This technique's drawback is that it is not starvation-free. A steady stream of processes changing the pointer, at the rate of one per memory cycle, can prevent the termination condition from ever becoming true. The advantage is concurrency. Subject to the "one-access-at-a-time" limitation imposed by the MM hardware and any data dependencies implied by the code, any number of processes can traverse the pointer concurrently using repeated reading. Although the loop in this atomic action may not terminate, execution of the atomic action does not introduce the potential for starvation for other processes since all

SCHED operations issued by the atomic action are cancelled on each failing iteration.

The repeated read method may be also be useful in some computations in which the access set can be statically determined. An atomic action that computes a function that takes a long time to compute may be executed by first reading the input variables and computing the function before issuing a scheduling parop containing the SCHED operation on the output variable and confirming R operations on the input variables. By decreasing the time between the execution of the SCHED operation and the corresponding ASSIGN operation on the output variable, this approach should decrease the time that process reading that variable are delayed. A similar use for the repeated read method is in executing atomic actions that write a variable that has been identified as a bottleneck.

THE RW-LOCKING TECHNIQUE. A third technique for executing atomic actions with dynamic access sets is RW-locking, intended for use with access variables that are pointers. The method is called RW-locking because the pointer functions as a lock that is set by scheduling a read access immediately followed by a write. The unsubstantiated write on the pointer creates a data dependency that delays all subsequently scheduled reads of the pointer. The pointer functions as a lock protecting the variable to which it points from access through that pointer until the process that set the lock releases it by substantiating or cancelling its write reservation on the pointer. The protection is partial — RW-locking a pointer does not prevent accesses by all other processes to the variable to which it points, only accesses through that pointer — but it is sufficiently powerful in combination with parallel operations on access sequences to support correct execution of atomic actions that read and dereference a pointer.

A process executes an atomic action using RW-locking in four steps:

(1) request the lock (or locks) by issuing an R-SCHED operation pair on the pointer.

(2) wait until the read to the pointer is substantiated, i.e., until the lock is acquired.

(3) issue a parop scheduling every other access required by the atomic action. (Note this parop is not a scheduling parop because it does not contain the R operation on the pointer.)

(4) release the lock by cancelling or substantiating the write reservation on the pointer. This operation can be issued in the parop issued in step two (unless the process needs a value read in the parop before it can release the lock) but must not be issued before the that parop.

The pointer example above is executed using RW-locking as follows:

```
R(c^.next,-,n) || SCHED(c^.next,pos);      set lock
W(n^.data,e) || CANCEL(c^.next,pos)        access data and release lock
```

Note that the CANCEL in this case is issued in the same parop that contains the operation dereferencing the pointer.

If any process uses a variable, V, as a lock, then every access to V must be consistent with V's function as a lock. In particular, the following restrictions apply to accesses to V:

(1) no process can write V without first reading V in the same atomic action. Blind writes have the effect of releasing a lock set by another process.

(2) Every process that accesses both V and the data protected by V in the same atomic action must first acquire the lock represented by V before scheduling any access to V^, Note that RW-locking and the repeated read technique can not both be used with the same variable. pp Atomic actions executed using the RW-locking method are correct because these restrictions, together with the sequential consistency of parops and the write-substantiation rule, imply a partial order over the atomic actions. Either for every variable accessed by both atomic actions A and B, A's access is scheduled before B's access or, for every such variable, B's access is scheduled before A's. Note that the partial order over parops implied by the sequential consistency of parops is insufficient to establish atomicity of atomic actions that use RW-locking because there is no scheduling parop — the pointer is read and dereferenced in separate parops. The restrictions above ensure that for any pair of atomic actions, A and B, that read and dereference the same pointer, V, using RW-locking, if A's R-SCHED operation on V is executed before B's, then A's operations scheduling accesses to V^ will also be executed before B's. Atomic action A does not substantiate or cancel its write

reservation for v until it schedules all of its accesses dereferencing v and B can not schedule any operation dereferencing v until A substantiates or cancels its write reservation.

Atomic actions executed using the RW-locking technique may violate the second restriction of the write-substantiation rule. For example, the atomic action

```
{n := c^.next; c^.next := n^.next }
```

executed using the RW-locking technique as follows:

```
R(c^.next,-,n) || SCHED(c^.next,pos);
R(n^.next,-,p);
ASSIGN(c^.next,pos,p);
```

violates the write-substantiation rule by assigning to `c^.next` a value read in a parop issued after the parop that schedules the write to `c^.next`.

Atomic actions executed with the RW-locking technique can deadlock if they fail to conform with the write substantiation rule and they operate on a data structure that contains a cycle. On a circularly linked list of *n* elements, *n* processes will deadlock executing the code above if `c` points to a different element for each process and the processes concurrently issue the first parop. Each process waits for its successor in the cycle of waiting processes to substantiate the link field in the element, `n`, that is the successor of its starting element, `c`. Note that a cycle in the structure is necessary in order to create a cycle of waiting processes. Atomic actions that use the RW-locking technique must be independently proven deadlock-free by showing that the structure accessed does not contain a cycle or that at least one element in the cycle will not be accessed concurrently with the other elements.

4.9. Examples

For each example problem, we present a solution written in a Pascal-style high-level language followed by the translation of that solution into a sequence of parallel operations on access sequences. Ideally the programmer writes the solution using a high-level language and the translation is done automatically.

COUNTER INCREMENT. Each process increments a shared counter, `V`, by the value of `delta`, a variable local to the process. The counter must be read and incremented atomically to prevent the erroneous answer that results if a process reads the counter between the time another process reads and writes the counter.

The counter increment

```
{V := V+delta}
```

is executed using parallel operations on access sequences as follows:

```
< R(V,-,var) || SCHED(V,pos) >;
ASSIGN(V,pos,var+delta)
```

If two processes concurrently increment the counter, the accesses are serialized in the order in which their scheduling parops arrive at the MM containing the counter. The increment is atomic because the scheduling parop for the increment reserves two contiguous elements in the counter's access sequence — a read immediately followed by a write. No other process can intervene by accessing the counter between the read and write.

The usual solution

```
repeat                                          busywait
until test-and-set(lock) = 0
var := V;
V := var+delta;
lock := 0;                                      release the lock
```

uses locks to ensure that the increment is executed atomically.

DELETION FROM A DOUBLY-LINKED LIST. Each process begins at the head of the list and searches for an element with a data field that matches `target`, a variable local to the process. If it finds a matching element, the process deletes it from the list and exits. We assume that the first element on the list is a sentinel element with a data field equal to `nilval`, a value that does not match any process's target. The shared variable `head` points to the sentinel element. The value of `head` does not change. Each element contains two link fields, the `f` (forward) link and the `b` (back) link.

The accesses required to delete an element must be done atomically or portions of the list can be lost. When a process deletes an element it changes the data value to `nilval` to prevent any other process with the same target that may be currently visiting the element from attempting to delete the element again. The data field is read twice to avoid the overhead of the atomic action if the data value is not equal to target. The second reading is necessary to ensure that no process deleted the element, changing its data value to `nilval` between readings. The procedure makes no provision for garbage collection.

```
p := head; found := FALSE;
REPEAT
   p := p^.f;
   IF p = nil THEN EXIT;
   IF p^.data = target
      THEN {IF p^.data = target
               THEN BEGIN
                    p^.b^.f := p^.f;
                    p^.f^.b := p^.b;
                    p^.data := nilval;
                    found := TRUE;
                    END; }
UNTIL found;
```

This search and delete procedure is executed using the repeated read technique as follows:

```
p := head; found := FALSE;
REPEAT
   R(p^.f,-,p);
   IF p = nil THEN EXIT;
   R(p^.data,-,d)
   IF d = target
      THEN BEGIN
           R(p^.f,-,f) || R(p^.b,-,b);                    the first reads
           done := FALSE;
           REPEAT
              R(p^.b,-,b') || R(p^.f,-,f') ||             the confirming reads
              R(p^.data,-,d) || SCHED(p^.data,dpos) ||
              SCHED(p^.f^.b,fpos) || SCHED(p^.b^.f,bpos);
              IF d <> target
                 THEN BEGIN
                      done := TRUE;
                      CANCEL(p^.f^.b,fpos) || CANCEL(p^.b^.f,bpos) ||
                      CANCEL(p^.data,dpos)
                      END
                 ELSE IF (b=b') AND (f=f')
                         THEN BEGIN
                              done := TRUE; found := TRUE;
                              ASSIGN(p^.f^.b,fpos,b) ||
                              ASSIGN(p^.b^.f,bpos,f) ||
                              ASSIGN(p^.data,dpos,nilval);
                              END
                         ELSE CANCEL(p^.f^.b,fpos) || CANCEL(p^.b^.f,bpos) ||
                              CANCEL(p^.data,dpos) ||
                              R(p^.f,-,f) || R(p^.b,-,b);   first reads for next iteration
           UNTIL done;
```

```
            END;
UNTIL found;
```

Though lengthy, the translation is essentially mechanical. The deletion is executed atomically because the reads for `p^.f`, `p^.b`, and `p^.data` and the writes for `p^.b^.f`, `p^.f^.b` and `p^.data` are all scheduled in the same parop. Note that the RW-locking technique cannot be used here. The double links create a cycle of pointers between pairs of adjacent elements.

DELETION FROM A SINGLY-LINKED LIST. The same problem — search and deletion of an element matching `target` — can be solved using the RW-locking technique on a singly-linked list. In the first solution below, variables are represented as single values and explicit use is made of locks. Each link field has an associated lock. When a process deletes an alement, `c`, it holds locks on `c`'s link field and on the link field of `c`'s predecessor. Since no other process can access either link, the deletion is atomic.

```
found := FALSE; g := head;
lock(g^.next);
p := g^.next;
IF p = nil THEN BEGIN unlock(g^.next); EXIT; END;
lock(p^.next);
c := p^.next;
IF p^.data = target
   THEN BEGIN
        g^.next := c
        unlock(g^.next);
        found := TRUE
        END;
WHILE (NOT found) AND (c <> nil) DO
   BEGIN
   unlock(g^.next);
   lock(c^.next);
   n := c^.next;
   IF c^.data = target
      THEN BEGIN
           p^.next := n;
           unlock(p^.next);
           found := TRUE;
           END
      ELSE BEGIN
           unlock(g^.next);
           g := p; p := c; c := n;          move to next element
           END;
    END;
IF NOT found                                unsuccessful search
   THEN BEGIN unlock(p^.next); unlock(g^.next) END;
```

Translation of a program that uses edge (pointer) locks into a program that uses the RW-locking technique is straightforward. The procedure above is executed using RW-locking as follows:

```
found := FALSE; g := head;
R(g^.next,-,p) || SCHED(g^.next,gpos);
IF p = nil THEN BEGIN CANCEL(g^.next,gpos); EXIT; END;
R(p^.data,-,d) || R(p^.next,-,c) || SCHED(p^.next,ppos);
IF d = target
   THEN BEGIN
        ASSIGN(p^.next,ppos,c);
        found := TRUE
        END;
WHILE (NOT found) AND (c <> nil) DO
   BEGIN
   CANCEL(g^.next,gpos) || R(c^.data,-,d) || R(c^.next,-,n) || SCHED(c^.next,cpos);
   IF d = target
      THEN BEGIN
           ASSIGN(p^.next,ppos,n);
           found := TRUE;
           END
      ELSE BEGIN
           g := p; p := c; c := n;          move to next element
           gpos := ppos; ppos := cpos;
           END;
    END;
IF NOT found                                unsuccessful search
   THEN CANCEL(p^.next,ppos) || CANCEL(g^.next,gpos)
```

Deletion is atomic because at the time of the deletion, the deleting process holds RW-locks on the link field of both the deleted element and the predecessor of the deleted element.

ASYNCHRONOUS GAME OF LIFE. Assume that the game is played on a two-dimensional array, `A[1..maxrow,1..maxcol]` by `maxrow * maxcol` processes. Process P_i is assigned to cell `A[(i div maxcol)+1,i mod maxcol]`. Each process repeatedly computes a new value for its cell that is a function of the values of the cell and the eight neighboring cells. The code for process P_i is as follows:

```
r := (i div maxcol)+1; c := i mod maxcol;
REPEAT
  { A[r,c] := f(A[r-1,c-1],A[r-1,c],A[r-1,c+1],
                A[r,c-1],  A[r,c],  A[r,c+1],
                A[r+1,c-1],A[r+1,c],A[r+1,c+1]) }
FOREVER
```

This atomic action is executed using parallel operations on access sequences as follows:

```
r := (i div maxcol)+1; c := i mod maxcol;
REPEAT
  R(A[r-1,c-1],-,nw) ||          R(A[r-1,c],-,n)          || R(A[r-1,c+1],-,ne) ||
  R(A[r,c-1],-,w) || R(A[r,c],-,me) || SCHED(A[r,c],pos) || R(A[r,c+1],-,e) ||
  R(A[r+1,c-1],-,sw) ||          R(A[r+1,c],-,s)          || R(A[r+1,c+1])-,se);
ASSIGN(A[r,c],pos,f(nw,n,ne,w,me,e,sw,s,se));
FOREVER
```

Since the reads for all the input values and the write for the output value are all scheduled in the same parop, execution is atomic. We expect the same approach to be useful in a number of more serious applications, including production systems and simulated annealing, where the basis for a process's action can be undermined by the action of other processes unless the basis is examined and acted upon as an atomic action.

5. SYNCHRONIZATION

Asynchronous shared memory computations require not only the ability to access multiple shared variables without interference, but also the ability to synchronize to ensure that execution respects the read-write, write-write, and write-read data dependencies implied by the code. These dependencies typically arise in the parallel execution of loops where a process must access the value of a variable accessed by another process in a previous iteration. A write-read dependency, for instance, is created when one process reads a variable that another process writes in a previous iteration. Synchronization between the two processes is required in order to ensure that the read occurs after the write. Locks are again the usual means of achieving the necessary coordination. The lock prevents the second access from occurring until the first access is complete. The locks can be on individual variables or on code. An example of a synchronization lock on code is the lock used in barrier synchronization. In barrier synchronization, no process can begin the next iteration until the last process to complete the current iteration releases the lock set at the beginning of each iteration.

Operations on access sequences provide a simple and flexible alternative. Synch operations can be used to control the order in which accesses are executed. When a process assigns a new value to a shared variable, `V`, it can ensure that a given other process reads the new value of `V` by issuing a parop containing a `W` operation immediately succeeded by an `R` synch operation, where the synch operation names the process that is to receive the new value. By scheduling more that one `R` synch operation, the process can send the value to more than one other process.

A simple example (that is not simple using traditional means of implementing synchronization) is a Jacobi iteration, an iterative method for solving linear equations. In a Jacobi iteration each process updates the value for the variable it has been assigned to compute with a value that is a function of the values of all the variables from the last iteration. The procedure below is for process P_1 in a Jacobi iteration involving three processes. In each iteration, each process exchanges the value of the variable it has been assigned to compute with the other two processes.

```
P1::  myval := initialvalue;
      REPEAT
         < W(V1,myval) || R(V1,-,-,P2) || R(V1,-,-,P3) >
         myval := f(myval, receive(V2), receive(V3))
      UNTIL done;
```

The `receive` construct sets up a local variable or register to receive the value from a read initiated by another process. A process executing a `receive` is delayed until the value has been received. The hyphen is a space holder. In the case of an `R` or `SCHED` synch operation the value corresponding to the second parameter is sent to the process specified in the operation, not the process that issued the operation.

An alternative way to enforce a write-read data dependency is for a process to schedule a write operation for another process immediately followed by a read access for itself:

```
P1::  < SCHED(V,-,P2) || R(V,-,var) >
P2::  ASSIGN(V,receivePos(V),val)
```

A process executing `receivePos(V)` is delayed until it receives from main memory a value for variable `V` that represents the position of the element in `V`'s access sequence to which the process should write.

The send construct enforces a write-read data dependency. The other data dependencies can be enforced in a similar way. A read-write dependency is created when a process must read the value of a shared variable before it is overwritten by another process. This can be implemented in either of two ways:

(1) "I read, you write". A process, P_2, that is reading a shared variable, `V`, can ensure that it sees the value of `V` before it is written by a given other process, P_2, by issuing an `R-SCHED` operation pair, where

the SCHED operation is a synch operation that names P_2.

```
P1::  < R(V,-,var) || SCHED(V,-,P2) >
P2::  ASSIGN(V,receivePos(V),val)
```

(2) "You read, I write". Alternatively, the process, P_2, that is writing the new value can schedule a read by P_1 to precede its write.

```
P1::  var := receive(V)
P2::  < R(V,-,-,P1) || W(V,val) >
```

The matrix transpose program below illustrates the use of the first of these two techniques for enforcing read-write data dependencies. The program uses *n* processes to transpose an *n*n* matrix. Process P_k copies column *k* to row *k*.

```
Pk :: FOR i := 1 TO n DO
        BEGIN
        < R(A[i,k],-,var) || SCHED(A[i,k],-,Pi) >;
        ASSIGN(A[k,i],receivePos(A[k,i]),var)          A[k,i] := A[i,k]
        END;
```

Each element A[i,j] of the matrix is read by process P_j before it is written by process P_i.

A write-write data dependency is created when two processes write a variable where one write logically succeeds the other. The specified order can be enforced by having one process schedule both writes, one for itself and one for another process.

These techniques for using operations on access sequences to enforce data dependencies in a shared memory computation can also serve as the basis for an implementation of a message based model (MBM) computation on a shared memory architecture. The technique used by a writer to enforce "I write, you read" synchronization can be used in the obvious way to implement send/receive communication, as follows:

```
send(V,val,P1,P2,...,Pn)  ≡
   < W(V,val) || R(V,-,-,P1) || R(V,-,-,P2) || ... || R(V,-,-,Pn) >
```

Other synchronization techniques using operations on access sequences are also useful. For example, a server/client interaction can be implemented using the "you write, I read" form of synchronization as fol-

lows:

```
client::  send(V1,argument,server) || < SCHED(V2,-,server) || R(V2,-,result) >;
server::  invalue := receive(V1);          read the input from the client
          ans := f(invalue);
          ASSIGN(V2,receivepos(V2),ans);   write the answer in the space reserved by the client
```

Note that the use of "you write, I read" synchronization permits the server to communicate with its clients without knowing their identities.

Access sequences provide a simple and elegant way to implement buffers. On a machine that represents variables as access sequences, most arrays will be represented as arrays of access sequences, but an array that serves as a buffer can be represented using a single access sequence.

```
Producer(s)::
   REPEAT
     produce_val(val);          produce a value, val, for the consumer
     send(V,val,consumer);
   FOREVER;

Consumer::
   REPEAT
     consume_val(receive(V));   receive and use the next value
   FOREVER.
```

Note that more than one process can add values to the buffer. A buffer for one producer and many consumers can be implemented as the access sequence of a single variable by using the alternative way of enforcing write-read data dependencies. Each consumer repeatedly schedules a write for the producer followed, in the same parop, by a read for itself. The operations we have defined on access sequences are not powerful enough to implement a buffer with many producers and many consumers using a single access sequence. The buffer must be implemented as usual as an array. In section 8 we propose additional operations on access sequences that support the single access sequence implementation.

Synch operations support SIMD-style programming. The following parallel-prefix computation on an array, `A[1..n]`, assigns to `A[i].sum`, for all `i`, the sum

$$\sum_{k=1}^{i} A[k].val$$

in $O(\log n)$ steps. Process P_i computes the partial sum for element `A[i]`.

```
P_i :: R(A[i].val,-,v);
      FOR k := 0 TO ⌈logn⌉ - 1 DO
         BEGIN
         IF i + 2^k ≤ n
            THEN send(A[i].sum,v,P(i + 2^k));
         IF i - 2^k ≥ 1
            THEN v := v + receive(A[i - 2^k].sum);
         END;
      W(A[i].sum,v);
```

On a typical asynchronous shared memory architecture, this computation requires barrier synchronization on each iteration to simulate the steps of the SIMD computation.

6. IMPLEMENTATION OF VARIABLES AS ACCESS SEQUENCES

The representation of variables as values requires adding intelligence to the memory modules. Instead of executing reads or writes on variables represented as single values, we require that the MMs execute `SCHED`, `ASSIGN`, `R`, `W`, and `CANCEL` operations on variables represented as access sequences. There are many alternative ways of implementing the access sequences and operations on access sequences. In this section we describe one alternative.

We implement the access sequence for variable `V` as an array, `V[1...] of V_record_type`, in main memory. One field of `V_record_type` is a value of `V_type`, where `V_type` is the declared type of `V`. The other field is a control field used to indicate whether the value recorded in the first field is an assigned value or one of the special values `TBA` for an unsubstantiated write, `TBA(pId)` for an unsubstantiated read, or `DELETED`. (The first field can be used to store the pId of the scheduling process if the size of the value field is always large enough to represent the highest pId.) For now, assume that the arrays representing access sequences are infinite. The MM maintains, for each array, an index which indicates the element in the array of the last element of the sequence. The value of the index for each array is initially 0 if the lower bound of each array is 1. Given this array implementation, the MM's execute each of the five defined operations as follows:

SCHED(V,pos,pID)

Increment the index for `V` and send to process `pId` `V`'s address and the value of `V`'s index.

ASSIGN(V,pos,val)

Assign the value `val` to element `V[pos]` and propagate the value to the succeeding reads. The

MM propagates the value by copying it to each element that follows `V[pos]` in the array until it has propagated the value through the last element of the sequence or reached an element added by a write (whether substantiated or not). Propagation continues through elements marked `DELETED`. Every time it copies the value to an element with value `TBA(a_pId)`, the MM also sends the value to process `a_pId`.

R(V,var,pId)

If `V[index]` has a special value (`TBA`, `TBA(a_pId)`, or `DELETED`), then the previous read is unsubstantiated. Increment index, assign to `V[index]` the value `TBA(pId)` and send process `pId` the address of `V` and the value of the index. Otherwise, send to process `pId` the address of `V`, value of `V[index]`, and a special value -1 as the pos parameter to indicate that the value returned is from a read access that was immediately substantiated.

W(V,val)

Increment index and assign the value `val` to `V[index]`. Note that no propagation can occur.

CANCEL(V,pos)

Change the value of `V[pos]` from `TBA` to `DELETED`. If the preceding write is substantiated, propagate the value of that write to reads succeeding the cancelled element.

The MM does not need to store the entire access sequence for a variable. The array for a variable need only hold the "active" portion of the access sequence, the subsequence beginning with the element preceding the first unsubstantiated write and ending with the "last" element of the sequence, `V[index]`. The implementation must either provide for overflow or ensure that the space allocated for the array is at least as large as the maximum size the active subsequence can attain during the computation. The MM can handle overflow locally (allocating another array in MM and linking it to the initial array) or by writing page sized chunks of a lengthy access sequence to secondary memory. The latter proposal assumes an architecture in which the MM's can access secondary memory quickly. Further research is needed to determine the best size for access sequence arrays and the best policy for handling overflow. We also intend to explore the effect of limiting each of the n process to c outstanding accesses, i.e., limiting the number of `R` and `SCHED` operations a process can issue for which there has been no corresponding assignment step. (For synch operations, enforcement of this limitation would require notifying the process that issued the operation when the assignment step is completed.) The limitation can be enforced at the process network interface and allows all the access sequences stored in a given MM to be represented as paths in an $n*c$ array in the MM.

Implementation of parallel operations on access sequences also requires modifying the processor network interface to provide mechanisms for receiving values returned from the MMs, notifying the

correct resident process of the receipt, and storing the value until it is retrieved by the process.

7. RELATED WORK

Our concept of representing variables as access sequences is similar to the use of multiple versions in implementing atomic actions as described by Reed [Ree78, Ree83]. There are two major differences. First, Reed's proposal does not rely on scheduling accesses. An advantage is that data dependent accesses (e.g., through pointers) can be handled in the same way as other accesses. A disadvantage is that the atomic action may be forced to roll-back. The second difference is in the underlying mechanism used to ensure atomic execution and the consequences of that difference. Reed's proposal for multiversion concurrency control uses timestamps instead of scheduling parops to specify the logical order in which access operations are to be executed. Unfortunately, access operations can arrive out of order, i.e., a write can arrive after a read that has a higher timestamp and thus logically follows the write. In contrast, when parops are used to schedule accesses, the order in which the scheduling access operations arrive defines a logical ordering of the operations that is consistent with atomic execution. One consequence of the fact that the timestamped access operations can arrive out of order, is that the sequence of versions must be implemented as a linked list (unless writes with a timestamp less than the timestamp of the latest version are aborted) instead of the more efficient array representation. A more significant consequence is that a tardy write, i.e., a write arriving after a read with a higher timestamp, must be aborted, resulting in the roll-back of the atomic action that issued the write.

Fujimoto has proposed implementing a scheme similar to Reed's in hardware [Fuj89].

8. CONCLUDING REMARKS

We have proposed a new method for coordinating access to shared memory in an asynchronous parallel computation. The method is based on the representation of shared variables as access sequences, the division of accesses into a scheduling and an assignment step, and parallel operations. The use of parallel operations on access sequences to coordinate access to shared memory has several advantages

over other coordination mechanisms because it does not rely on locks. We do not claim these advantages for programs that use RW-locking or that use parallel operations on access sequences to simulate locks. The advantages include the following:

(1) The timing of reads and writes to shared variables is controlled by data flow, the availability of the values to be read or written, not by locks. Writes can occur in any order. They are not constrained to occur in the logical order defined by the order in which the writes are scheduled. Writes never wait, except for completion of the scheduling step. Once the value to be written has been computed the write can be substantiated without waiting for logically preceding reads to complete. A write is required to wait if the value to be written is computed after the `SCHED` operation for the write is issued but before the `SCHED` operation returns the index of the position reserved for the write. A read waits only if the preceding write is unsubstantiated. If the preceding write is substantiated, the read can be executed even if earlier writes, scheduled before the preceding write, are still unsubstantiated.

(2) No space is needed for locks or for process queues associated with locks. This advantage helps offset, but does not eliminate, a disadvantage of the use of parallel operations on access sequences — the increased space necessary for the access sequence representation of variables.

(3) Reads and writes execute without intervening lock and unlock operations. Processes do not need to set and release locks and any number of both read and write accesses to the same shared variable can be scheduled or substantiated in a single memory cycle, that is, without intervening communication across the ICN. For example, if `V`'s access sequence, starting with the first unsubstantiated write is `S,R(P1),R(P2),S,R(P3)` and the `ASSIGN` operations corresponding to both scheduled writes arrive in the current cycle, then the MM can immediately substantiate both writes and all the reads. In a system based on locks, execution of the accesses would be interleaved with lock and unlock operations, each requiring one or more trips across the ICN. Process P_1's read could not be executed until after the process that executes the first write in the sequence releases the lock on `V`

and P_1 acquires the lock. The read by P_3 could not be executed until V's lock was acquired or released eight times.

(4) Variables are always available for access, though substantiation of the accesses may be delayed by the inavailability of data. In contrast, in a system based on locking, a variable is unavailable not only during the time the process holding the lock needs exclusive access to the variable, but also for the time it takes to confirm and release the lock and obtain other locks needed for the atomic action. In order to avoid deadlock, a process typically is constrained to acquire locks sequentially, waiting for confirmation of each lock before attempting to acquire the next, so the total time the variable is unavailable may be significantly larger than the time the process actually needs exclusive access to the variable.

(5) Implementation of atomic actions is deadlock-free.

(6) Access to shared variables is fair. Accesses are scheduled in FIFO order and executed as soon as the value to be written or read is available.

(7) Atomic actions can be pipelined. After scheduling an atomic action, the process can schedule another atomic action without waiting for the accesses scheduled in the current atomic action to be substantiated. Pipelining accesses can help mask the memory access latency. Theoretically, with an ICN of sufficiently high bandwidth, a process can complete atomic actions at the same rate at which it can assemble and accept communication to and from the MM's and perform the local computation required for the atomic actions, i.e., at a rate independent of the network latency. In practice, pipelining will be limited to a degree dependent on the application by the occurrence of dynamically determined access sets. An implementation designed to bound the size of the access sequences may also limit the length of the pipeline by limiting the number of scheduled but unsubstantiated accesses that a process can have outstanding. For systems based on locking, pipelining atomic actions decreases concurrency and increases the potential for deadlock.

(8) A speculative advantage for parallel operation on access sequences is that such operations may be combinable. (Combining is briefly discussed below.) Accesses to locked variables definitely are not combinable.

The disadvantages to parallel operations on access sequences are the increased cost and complexity of the ICN and MM's, the need for the process (or process network interface) to keep track of unsubstantiated reads, the additional space required by the access sequence representation, and a longer memory cycle time implied by the increased complexity of the ICN and MM's. Whether parallel operations on access sequences is a practical method for coordinating access to shared memory will depend on the cost of the additional hardware and on the results of the tradeoff of an increase in memory latency for a decrease in the effect of the latency.

There are many aspects of our proposal that we have yet to fully explore. A list of some of these topics follows, along with a brief and tentative discussion of each and an indication of the approach we plan to take or the expected results:

• **Fault-detection and recovery**. The system we propose may support fault-detection and recovery as well as coordination of access to shared memory. In particular, the heartbeat of pulses generated by the PE's and propagated through the ICN to support parops may also be useful in detecting failures in the PE's and ICN. The access sequences resemble logs and may be useful in fault recovery.

• **Combining**. Some parallel architectures allow certain operations to be combined in the ICN, increasing the potential concurrency of access to shared memory [KRS88]. Since parallel operations are compatible with combining [RWW89], we believe that some combining of the operations defined here may also be possible. For instance, a switch in the ICN could combine a group R operations on the same shared variable that arrive at the switch without other operations on that same variable intervening, forwarding only a single R operation from the group, and satisfying all the combined reads with the value received back from the MM.

• **Sequential programming.** In most sequential programs the representation of variables as sequences would be merely burdensome. The last value would always be the relevant value. But in sequential programs which are in some sense "about" time, for instance simulations or real-time programs, the access sequence representation may be useful.

• **Additional operations on access sequences.** We intend to consider expanding the set of operations defined for access sequences. Candidate operations include the following:

— a ''fastread'' operation. A fastread operation immediately returns a recent value for the variable. If the last write to be scheduled is unsubstantiated, the operation returns the value assigned by the most recently scheduled substantiated write, even if the most recently scheduled write is unsubstantiated. The fastread operation could be useful in real-time computations and relaxed algorithms.

— acknowledged `R` and `SCHED` synch operations. When the assignment step for a synch operation is executed, the MM notifies the process that scheduled the synch operation. This acknowledgement is necessary for any scheme for limiting the size of access sequences based on limiting the number of unsubstantiated accesses a process can schedule.

— read-any and write-any operations. Instead of naming a specific process, a synch operation can specify that any process can execute the assignment step. For example `SCHED(V,pos,ANY)` schedules a write access where the process that executes the assignment step is determined dynamically. The operations are necessary when the name of the other process is unknown to the scheduling process and is useful for resource allocation and many-to-many synchronization.

A `W` operation is ''matched'' to the oldest `SCHED(ANY)` element in the access sequence, if there is one. Otherwise it is executed as before, by appending an element to the access sequence. Similarly, an `R` operation is matched to the oldest `R(ANY)` element in the access sequence. Matching a `W` operation to a `SCHED(ANY)` element means assigning the value specified to that element and propagating the value. Matching an `R` operation to an `R(ANY)` element means recording the `pId` of the process that issued the `R` operation in the element so that when a value propagates through the element it will be the value will be

sent to that process.

Given these operations, we can define special synchronization variables on which the only legal operations are `SCHED(ANY)-R` and `W-R(ANY)` operation pairs. A `SCHED(ANY)-R` operation pair is executed by "matching" the `R` operation with the oldest `R(ANY)` operation in the access sequence, if there is one. If there is no `R(ANY)` element in the access sequence, then a `SCHED(ANY)` element is added to the access sequence followed by an element reserving the read for the process that issued the operation pair. A `W-R(ANY)` operation pair is executed by matching the `W` operation with the oldest `SCHED(ANY)` element. If there is no unsubstantiated write, an element recording the value written is added to the access sequence followed by an `R(ANY)` element.

A buffer accessed by many producers and many consumers can be implemented using a single such synchronization variable, `V`, as follows:

```
Producers ::
   REPEAT
     produce_val(val);
     < W(V,val) || R(V,-,-,ANY) >;          I write - any process reads
   FOREVER;

Consumers ::
   REPEAT
     < SCHED(V,-,ANY) || R(V,-,val) >;      any process writes - I read
     consume(val);
   FOREVER.
```

Each value written by a producer is read by exactly one consumer in FIFO order.

— a read-all operation. A `W-R(ALL)` operation pair changes the value of a variable and broadcasts the new value to all processes. Efficient implementation requires adding intelligence to the ICN so that it can fan-out values sent from memory to all the PEs.

• **Production systems.** An impediment to parallel execution of rule production systems is the fact that the application of one rule may interfere with the application of another rule. To avoid this interference, processes synchronize after identifying the set of rules that are eligible for firing in order to select the single rule or the set of noninterfering rules to fire. This production cycle severely limits the speedup obtainable by parallel execution of production programs. A study of existing production systems shows

that rule parallelism is of limited effectiveness because firing a rule typically affects only a small constant number of other rules [GFN89]. Processes assigned to unaffected rules must either wait for the processes whose rules were affected by the rule fired in the last cycle or must participate in reevaluating the affected rules at a fine-grained level of parallelism.

An approach suggested by our proposal for coordinating access to shared memory is to rely on rule parallelism but eliminate the synchronization point in the production cycle. Each process would fire its rule whenever the rule became applicable. Processes executing the production program would use the repeated read technique. As soon as a process determines that its rule is eligible for firing, it would, in a single parop, reread the facts justifying firing and schedule writes for the facts that are changed by firing. If the second reading shows that firing is still justified, the assignments for the scheduled writes would be executed, otherwise, the writes would be cancelled. Instead of limiting the amount of parallelism, the fact that firing a rule affects only a few other rules is actually a benefit since it implies minimal interference — the facts justifying firing will rarely change between readings. For this reason we expect that the level of parallelism achievable will be on the order of the number of rules in the production system. A limitation imposed by this approach is that correctness of the production program must not depend on any given policy for selecting a rule to fire from among the rules eligible for firing.

• **Message-based model computations.** In the message-based model (MBM), there are no shared variables — a process can access only the variables in its local memory. Nevertheless, our proposal for coordinating access to shared variables may have a dual in the MBM based on a correspondence between MMs in the shared memory model and processes, as mediators of access to local variables, in the MBM. We would expect that our proposal, mapped to the MBM, would allow a process to schedule actions by several different processes that appear, to every other process in the computation, to occur as an indivisible operation. For object-oriented computations, this capability would allow an object to be distributed over several different processes without requiring locking to coordinate the activities of the constituent processes. The greater processing power associated with memory in the MBM may make a wider range of operations on access sequences possible.

• **High level constructs.** The access sequence operations defined in section 3 are not intended for direct use by the programmer. We intend that the programmer use higher level constructs that are compiled into these operations. For atomic actions, the translation into parallel operations on access sequences is straightforward unless a pointer or array index read within the atomic action determines the access sets. An initial question is whether automatic translation of such atomic actions is possible given the choice of execution techniques. For synchronization, we have proposed using send and receive statements for "I write, you read" synchronization. High level constructs to express other synchronization patterns are needed.

In a high level language based on parallel operations on access sequences provision should be made for declaring special variables on which only certain operations are legal, e.g. pointers on which RW-locking is used, so that the compiler can enforce the restriction. We also intend to investigate the extent to which variables represented as single values are compatible with variables represented as access sequences. Where there is a choice, the programmer should be able to declare the desired representation.

There are choices that need to be explored in defining the semantics of the operations where the basic mechanism we have proposed can support more than one alternative. For example, are accesses blocking? That is, does process P_1 wait after it issues an `R(V,var,1)` operation until it is receives the value of `V`?

• **Performance evaluation.** A pressing problem is how to evaluate the performance parallel operations on access sequences. Comparisons based on the number of global references are biased in favor of our proposal because the memory cycle time will be longer for the machine we have described than for a typical shared memory multiprocessor. We expect that any useful simulation will have to be finely detailed in order to capture the additional cost of supporting parallel operation on access sequences.

References

[Fuj89] R. M. Fujimoto, The Virtual Time Machine, *Proc. of the ACM Symp. on Parallel Algorithms and Architectures* , Santa Fe, New Mexico, June 18-21, 1989, 199-208.

[GFN89] A. Gupta, C. Forgy and A. Newell, High-Speed Implementations of Rule-Based Systems, *ACM TOCS 7*, 2 (May 1989), 119-146.

[KRS88] C. P. Kruskal, L. Rudolph and M. Snir, Efficient Synchronization on Multicomputers with Shared Memory, *ACM Trans. Prog. Lang. and Systems 10*, 4 (October 1988), 579-601.

[OwL82] S. Owicki and L. Lamport, Proving Liveness Properties of Concurrent Programs, *ACM Trans. Prog. Lang. and Systems 4*, 3 (July 1982), 455-495.

[Ree78] D. Reed, Naming and Synchronization in a Decentralized Computer System, MIT/LCS/Tech. Rep. 205, Laboratory for Computer Science, MIT, September 1978.

[Ree83] D. Reed, Implementing Atomic Actions on Decentralized Data, *ACM TOCS 1*, 1 (February, 1983), 3-23.

[RWW89] P. F. Reynolds, Jr., C. Williams and R. R. Wagner, Jr., Parallel Operations, Tech. Rep. 89-16, University of Virginia, Department of Computer Science, December, 1989.

[ShS88] D. Shasha and M. Snir, Efficient and Correct Execution of Parallel Programs that Share Memory, *ACM Trans. Prog. Lang. and Systems 10*, 4 (October, 1988), 282-312.