**Efficient Synchronization on
Multiprocessors with Shared Memory**

by

Clyde P. Kruskal†

Larry Rudolph∗

Marc Snir‡

Ultracomputer Note #105

Revised: March, 1988

† **Current Address:** Computer Science Dept., Inst. for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742

∗ **Current Address:** Institute of Mathematics and Computer Science The Hebrew University of Jerusalem, Jerusalem, Israel

‡ **Current Address:** IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598

Part of the work was done while the first author was at the University of Illinois, the second author was at Carnegie-Mellon University, and the third author was at Hebrew University and the Courant Institute of Mathematical Sciences, New York University.

A preliminary version of this paper appeared in the *5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1986.

ABSTRACT

A new formalism is given for read-modify-write (RMW) synchronization operations. This formalism is used to extend the memory reference combining mechanism, introduced in the NYU Ultracomputer, to arbitrary RMW operations. A formal correctness proof of this combining mechanism is given. General requirements for the practicality of combining are discussed. Combining is shown to be practical for many useful memory access operations. This includes memory updates of the form *mem_val := mem_val op val*, where op need not be associative, and a variety of synchronization primitives. The computation involved is shown to be closely related to parallel prefix evaluation.

1. INTRODUCTION

Shared memory provides convenient communication between processes in a tightly coupled multiprocessing system. Shared variables can be used for data sharing, information transfer between processes, and, in particular, for coordination and synchronization. Constructs such as the semaphore introduced by Dijkstra in [Di], and the many variants that followed, provide convenient solutions to many synchronization problems involving arbitrary number of processes. These constructs are supported in hardware by machine instructions that atomically execute a Read-Modify-Write cycle. Such instructions exist on most modern CPU's.

An atomic Read-Modify-Write operation only requires that it be semantically atomic, although it is often processed atomically also. The ''serial bottleneck'' created by this atomic processing, while acceptable for small scale parallelism, can seriously impair the performance of a system with thousands of processors.

Frequent accesses to a shared variable not only slow down those processes performing the access, but may cause the entire machine to thrash. Large-scale shared memory parallel processors are likely to use multistage packet switched interconnection networks for processor to memory traffic. These networks provide high bandwidth and short latency time when memory accesses are distributed randomly, but, if even a small percentage of the memory requests are directed to one specific spot, the network becomes congested and performance quickly degrades. A study of Pfister and Norton [PN] shows that not only those processors attempting to access the same ''hot spot'' are delayed, but also the remaining processors (see also [LKK], [RT]). Although replication of data can often be used to circumvent the hot spot problem for read-only data, it cannot be used for synchronization variables.

The performance degradation can be mitigated by a memory request ''combining'' technique. Briefly, combining works as follows: When a ''conflict'' occurs within the network for the same switch output port for memory requests directed to the same location, a new combined request that represents the conflicting requests is created. Separate replies to the original requests are later created from the reply to the combined request. The logic for combining and uncombining memory references is distributed throughout the processor-to-memory interconnection network.

It is worthwhile emphasizing that such simultaneous requests directed at the same memory cell are not random, rare events. When processed in an efficient manner, they can form the basis for a completely parallel, decentralized operating system as well as a building block for efficient parallel programming constructs.

Indeed, such a combining mechanism was proposed for read requests in the CHoPP machine [SBK]. It was extended to handle write requests, and some types of Read-Modify-Write requests [Ru] and further generalized for associative Read-Modify-Write operations [GK]. These ideas are used to implement concurrent reads, writes, and ‘‘Fetch-and-Adds’’ in the NYU Ultracomputer [GGK] and IBM RP3 [PBH] machines.

It is relatively easy to argue on the correctness of serial computers. These are relatively simple systems; and there is a well understood abstract model, hence clear correctness criteria. On the other hand, our intuition often fails when trying to reason correctly about complex, parallel systems. Thus, it is important to precisely define correctness criteria for parallel systems and to formally argue that these criteria are fulfilled. Work on the semantics of concurrent processes (e.g. [LyF], [La1], [La2], [La3]) have recently supplied the formal framework for such activity. Yet, there have been very few applications of this formalism to the design and analysis of parallel computer architectures.

We show in this paper that combining fulfills two important criteria: (1) Combining is a general technique that applies to arbitrary memory access operations, not just an *ad hoc* method to handle the NYU Ultracomputer operations. (2) This new interconnect mechanism does not change the properties of the memory system.

These two issues are addressed rigorously. A new, very general formalism for read-modify-write (RMW) operations is given. A general definition is given of a correct machine implementation. A method for combining general RMW operations is given and proven to be correct. Several families of memory access operations are analyzed using this general framework. This includes familiar operations such as load, store, swap, test-and-set, fetch-and-add, and general data-level synchronization primitives (see [GP]). It is well known that any associative operation can be combined efficiently [GK]. We show that other combinable families of operations include the four standard arithmetic operations, all sixteen boolean functions, and synchronization methods such as full/empty bits. Implementation issues concerning support of such primitives are considered. Finally, the combining mechanism is shown to be closely related to the parallel prefix computation problem [LaF].

2. READ-MODIFY-WRITE

We use a formalism similar to that developed by Lynch and Fisher [LyF]: A parallel computation consists of a set of processes that execute in parallel. Each of these processes is considered to be a sequential program augmented with the ability to access global, shared variables. We restrict our attention to shared memory access techniques and assume standard operations for manipulation of local (or private) data.

Instead of the usual load and store memory access operations of sequential processing, all accesses to shared variables are assumed to be *Read-Modify-Write (RMW)* operations. The operation $\mathrm{RMW}(X, f)$, where X is a shared variable and *f* is a mapping, is defined to be equivalent to the indivisible execution of the following function:

```
function RMW(X,f)
   begin
      temp ← X;
      X ← f(X);
      return(temp)
   end
```

This operation yields, as its value, the old value of the variable X and also updates the value stored in X according to the updating transformation *f*.

The usual load and store operations are particular cases of RMW operations: A *load* from (the address of) variable X is equivalent to RMW(X,**id**), where **id** is the identity mapping (i.e. $f(x) = x$). A *store* of value v to variable X is equivalent to RMW(X, $\mathbf{I}_v$), where $\mathbf{I}_v$ is the mapping that has constant value v (i.e. $f(x) = v$); the returned value is ignored. In fact, an assignment of the form $Y \leftarrow \mathrm{RMW}(X, \mathbf{I}_Y)$, where Y is a private variable and X is a shared variable, implements a *swap* operation: X and Y swap values. Note that the usual use of swap operations is to exchange values between a shared variable (the lock) and a private variable (the key) (see, e.g. [PET], §9.5.4).

The well known *test-and-set* operations can also be implemented as an RMW operation. We have

$$\text{test-and-set}(X) \equiv \mathrm{RMW}(X, \mathbf{I}_{\mathrm{true}}) .$$

A more powerful RMW operation is the *fetch-and-add* synchronization primitive. It is defined by

$$\text{fetch-and-add}(X,a) \equiv \mathrm{RMW}(X, +_a) ,$$

where $+_a$ is Curried addition, i.e. $+_a(x) = x+a$. It corresponds to the indivisible execution of the following code.

```
function fetch-and-add(X,a)
   begin
      temp ← X;
      X ← X+a;
      return(temp)
   end
```

The replace-add operation was introduced many years ago [DGSS]; it is identical to fetch-and-add, except that the updated value, rather than the old value, is returned. It was independently considered by Dijkstra [Di] who rejected it, believing it to be an inadequate tool for synchronization. It has nevertheless turned out to be a very useful synchronization primitive, and was essential in the development of efficient coordination code for the NYU Ultracomputer operating system [Ru], [GLR]. The change from replace-add to fetch-and-add [GK] simplified the combining logic and paved the way to the general result given in this paper.

Any memory access that consists of reading one shared memory location, performing an arbitrary local computation, then updating the memory location can be expressed as an RMW operation of the above form. This is the general form for memory accesses assumed by Lynch and Fisher, and seems to encompass most, if not all, useful synchronization operations based on shared variables. Other examples of RMW operations will be presented in Section 5.

The instruction set of most modern processors support such RMW operations. In the usual, ‘‘processor-side’’, implementation of RMW operations the updating is done by the processor, and communication with memory uses a ‘‘load-store’’ extended cycle:

- The processor issues a load;
- the old value is returned from memory;
- the processor computes an updated value;
- the processor issues a store;
- the updated value is stored in memory.

Three messages are exchanged between processor and memory (four if memory acknowledges each store); the memory itself is locked for the duration of this extended cycle, to prevent another access to the same location (this is often achieved by locking the memory bus).

An alternative, ‘‘memory-side’’, implementation is to have the update computed at the memory itself:

- The processor issues a RMW request (containing opcode, address, and data);
- the memory controller updates the content of the addressed location;
- the old value is returned from memory.

Only two messages are exchanged between processor and memory; and memory is locked only during the execution of the update operation.

The second implementation reduces the processor to memory traffic, and avoids long memory locking; however, it requires a more complex memory controller, that can execute locally ‘‘read-modify-write’’ cycles. Note that many memory controllers have this ability, in order to support a byte or half-word write: the full word is read, modified, and stored back.

The second implementation method seems much preferable in large shared-memory multiprocessors. It is used in the NYU Ultracomputer and IBM RP3 (which use combining), and in the BBN Butterfly [RT] and Univ. of Illinois Cedar [ZY] (which do not use combining).[1] We assume in the sequel

[1] Unfortunately, usual commercial processors do not have a machine instruction that executes an atomic ‘‘store-load’’ extended cycle. The NYU Ultracomputer, IBM RP3 machine, BBN Butterfly, et al. emulate it by using a store, followed by a load.

that RMW operations are implemented using the second method.

3. MEMORY MODELS

3.1. The Uniprocessor Memory

A computation on a uniprocessor system consists of the execution of a serial stream of instructions. The outcome of the computation is as if these instruction where executed sequentially in the order specified by the program; the execution of one instruction terminates before the execution of the next instruction starts. In practice the execution of successive instructions is often overlapped, in order to increase performance. This is especially important for memory accesses; pipelining of memory accesses can mask the (usually) high latency of memory, and balance memory throughput to processor speed. This overlapping should be invisible; the behavior of the computation is as if no overlapping occured.

A uniprocessor system consists of processor and memory. Processor and memory communicate by exchanging messages. For each memory access the processor sends a request message to memory and eventually receives back a reply message. (We assume each access generates a reply; this reply carries a value if the memory operation is a load or a read-modify-write; it carries an acknowledgement if the operation is a store.) The basic memory operation consists of receiving such a message, modifying the contents of memory if necessary, and sending back a reply. This operation is executed atomically; the memory behaves as if the operations were executed serially, in some order.

The processor respects obvious data dependencies in its interaction with memory; i.e. if it loads a value from memory then the operation using that value is not executed until the value is returned from memory. In addition, some constraints are needed on the order memory operations occur. For example, if the processor issues a store, followed by a load to the same location, then the store should take place first; otherwise the outcome would not be consistent with the assumption that instructions execute atomically, in the order specified by the program.

The simplest constraint is to assume that memory is a FIFO server:

(U1) Memory accesses are executed in the order requests are submitted by the processor.

This is the policy used in most memory systems.

A weaker constraint is to assume that each memory location is a FIFO server:

(U2) Memory accesses at each memory location are executed in the order the corresponding requests are issued by the processor.

For example, if the memory consists of several separate modules, then each module can have its own queue of requests; requests going to distinct modules may be executed out of order. It is easy to see that this constraint is sufficient to prevent memory hazards.

3.2. The Multiprocessor Memory

A multiprocessor consists of several processors and a memory shared by all processors (each processor may also have its own private memory). Processor to memory communication, and memory operations are as in the uniprocessor case. The usual model for such systems is provided by the *sequential consistency principle* [La1] (see also [LyF]):

> Each instruction is executed atomically, and instructions within the serial stream of each processor are executed in the order specified by the program of this processor; i.e. the outcome of the computation is as if all the instructions where executed sequentially, in a sequence obtained by interleaving the sequential streams of instructions executed by each processor.

This principle is about the visible behavior of the computation, not about the implementation. Obviously, the execution of instructions by distinct processors is expected to take place concurrently. We also expect to have intraprocessor overlapping of operations, and in particular pipelining of successive accesses by a processor to shared memory. The shared memory of a large multiprocessor has a relatively large latency (due to the complex interconnect mechanism, the need for arbitration, and possible conflicts). For large numbers of processors and memory modules, high bandwidth between processors and shared memory can be obtained only by such pipelining.

Again, some constraints are required on the order memory operations are executed. The simplest such constraint is to assume that the memory subsystem is a FIFO server:

(M1) The memory receives a sequential stream of requests from the processors; this stream is obtained by merging the serial streams of requests generated by individual processors. (The relative order of requests generated by the same processor is preserved; the relative order of requests generated by distinct processors is not significant.) The requests are processed in the order the appear in this stream.

This condition is sufficient to enforce sequential consistency [La1]. However, to do so requires a central memory controller, which may significantly limit memory bandwidth in a large scale parallel processor with hundreds of processors and memory modules.

We can consider a weaker constraint, similar to the weaker constraint (U2) for the uniprocessor case:

(M2) Each memory location receives a sequential stream of requests from processors; this stream is obtained by merging the serial streams of requests directed to that location by the individual processors. The requests are processed in the order they appear in the stream.

A statement equivalent to this condition is that successive requests submitted by a processor to the same memory location are processed in their order of submission.

To see where conditions (M1) and (M2) differ, consider the parallel execution of the following two instruction streams (the example is due to Collier [Co]).

Processor 1	*Processor 2*
(1) load A	(3) store B←1
(2) load B	(4) store A←1

To fulfill condition (M1), memory must behave as if accesses occur in one of the six orders in which (1) preceeds (2) and (3) preceeds (4):

1234, 1324, 1342, 3124, 3142, or 3124 .

Assume that initially A=B=0. For the first order, the loads will return 0 for A and B, and, for the remaining orders, the loads will return 0 for A and 1 for B. On the other hand, to fulfill the weaker condition (M2), accesses may (for this example) occur in any order. If accesses occur in the order 4123, the loads will return a value of 1 for A and a value of 0 for B, an outcome that is not sequentially consistent. Thus, condition (M2) is not sufficient to enforce sequential consistency.

Nevertheless, the memory subsystem of many shared memory parallel computers, in particular the NYU Ultracomputer and the IBM RP3 machine, only satisfy condition (M2). In order to enforce sequential consistency these machines rely on stronger processor control of memory accesses: an operation may wait until some previous memory access occurred, even if it is not data dependent on this access. Such delays can be used to prevent hazards due to *interdependencies*, as illustrated in the previous example. For example, the RP3 machine provides a *fence* instruction: the execution of a fence causes the processor to wait until all of its outstanding references to shared memory have terminated. An incorrect execution can be prevented in the previous example by adding a fence between the two memory accesses in each of the serial streams. Compile time analysis of possible hazards can be used to determine where such fences are needed [SS]. This results in minimal loss of interprocessor and intraprocessor concurrency in access to shared memory.

We henceforth assume that a memory subsystem is correct if each memory operation is atomic and condition (M2) is fulfilled. A more detailed definition of M2 is:

(M2.1) The behavior of the memory system is as if it executed a serial stream of atomic operations, each one consisting of accepting a processor request, processing that request, and returning a reply;

(M2.2) each request that arrives at the memory system is eventually accepted; and

(M2.3) successive requests sent by a processor to the same memory location are accepted in the order they enter the memory system.

We do not encompass timing constraints in our definition of correctness; a practical system will also have constraints on the time it takes to access memory, and one would need to prove that these constraints are satisfied.

A large, shared memory is constructed of several independent memory modules; an interconnection network is used to transfer requests from processors to memory and transfer back replies. Thus a memory system is actually composed of memory modules and an interconnection network. We must show that if the memory modules function correctly and the interconnection network functions correctly, then the memory system functions correctly.

A memory module functions correctly if it fulfills (M2.1)-(M2.3). We say that the interconnection network functions correctly if it fulfills the following two conditions:

(M2.4) Each message sent in the network eventually reaches its destination; and

(M2.5) successive messages sent from the same source to the same destination arrive in the order they were sent.

One can easily check that the following holds:

Lemma 3.1. Consider a memory system built of an interconnection network and independent memory modules. Assume that the interconnection network functions correctly (i.e. satisfies (M2.4),(M2.5)), and that each memory module functions correctly (i.e. satisfies (M2.1)-(M2.3)). Then the memory system functions correctly (i.e. satisfies (M2.1)-(M2.3)).

4. COMBINING MECHANISM

There have been many proposals for the architecture for parallel processors. The main issue is how to interconnect the processors so that they may communicate efficiently. While shared bus type architectures are well suited for interconnecting dozens of processors and memory modules, multistaged interconnection networks appear to be required for larger scaled parallel machines. We first describe our assumptions concerning the interconnection network and then give a general technique for ‘‘combining’’ shared memory requests directed at a common location. We show that this implementation is correct in the sense defined in the previous section.

4.1. Processor to Memory Connection

We assume a multiprocessor memory system built of memory modules and an interconnection network, where both the memory modules and the network function correctly. For the sake of definiteness, we make the following additional assumptions:

- The processors communicate with shared memory modules via a multistage interconnection network. The network is packet switched.
- The network is ‘‘nonovertaking’’: If message m_1 leaves some node before message m_2 and they both arrive later at some other node, then they arrive in the correct order. Successive messages sent by a processor to the same destination arrive at every node in their shared path in the order they were sent.
- A reply message is sent back on the same path followed by the request message.

The last two conditions are trivially satisfied for multistage networks that have a unique path connecting each processor to each memory module. The last condition is easy to enforce in any network: As a message travels through the network, it can construct a header describing its path; this header is used to route the reply in the reverse direction [GGK]. These assumptions are also made in the NYU Ultracomputer [GGK] and IBM's RP3 machine [PBH].

4.2. How to Combine Requests

We assume that memory accesses are RMW operations. A memory request message has the form $<id,addr,f>$, where id is an identifier that uniquely identifies the request, *addr* is a reference (address) to a memory location,[2] and *f* is (an encoding of) a mapping. Let $@addr$ represent the value contained in location addr. When the message reaches memory, the contents of location addr, is replaced by $f(@addr)$, and a message $<id,@addr>$ containing the *original* contents of location addr is returned.

Suppose that a request message of the form $<id_2,addr,g>$ arrives at a switch containing a request message $<id_1,addr,f>$. These two messages have the same destination and thus conflict. We propose *combining* these two messages into a single message. This is done as follows:

- The switch saves id_1, id_2, and *f* and forwards the message $<id_1,addr,f \circ g>$, where $f \circ g$ is (an encoding of) the composition[3] of *f* and g.
- When a reply message $<id_1,val>$ to this composed request reaches the switch, the saved information is retrieved by matching the id's; a message $<id_1,val>$ is forwarded as a reply to the first request $<id_1,addr,f>$, and a message $<id_2,f(val)>$ is forwarded as a reply to the second request $<id_2,addr,g>$.

Assume that the combined request $<id_1,addr,f \circ g>$ is not further combined in the network. Then $<id_1,@addr>$ is returned as a reply, and the value $@addr$ is replaced in memory by $g(f(@addr))$. At the switch the reply $<id_1,@addr>$ is forwarded (back) to the first request, and the reply $<id_2,f(@addr)>$ is forwarded (back) to the second request. This is illustrated in Figure 1. The final effect is as if the first request was executed, returning the value $@addr$ and replacing it in memory with $f(@addr)$, and then the second request was executed, returning the value $f(@addr)$ and replacing it with $g(f(@addr))$. Combining is transparent: the operations executed by the processors and the final memory content are the same as would occur without combining.

4.3. Correctness of Combining Mechanism

We now show that combining is correct: We consider a multiprocessor memory system consisting of a correct combining network and correct memory modules. We prove that the resulting memory system behaves correctly. To do so we show that the observable behavior of a combining memory system is a behavior that could be observed in a noncombining one. Note that the reverse is not necessarily true: there are sequences of events that can occur in a noncombining memory system, but can not occur in a

[2] The address may be part of the identifier. Thus, if each processor has at most one outstanding request to each address, then the processor number can be used as identifier.

[3] We use $f \circ g(x)$ to denote $g(f(x))$.

Figure 1 - Combining Switch

combining one. (We follow what Lamport calls the ‘‘restrictive’’ approach to specification [La3].)

In general, a combined request can be further combined. An inductive proof is needed to show that the final outcome is correct.

Each memory request message in the network is associated with a sequence of memory request messages issued by processors. A memory request issued by a processor *represents* itself; if memory request A was obtained by combining B with C, where B represents requests $b_1, \cdots, b_i$ and C represents requests $c_1, \cdots, c_j$, then we say A *represents* requests $b_1, \cdots, b_i, c_1, \cdots, c_j$.

For each request traversing the network, there is a unique return message; in particular, each processor request is associated with a unique reply returned from memory. More formally, for each request message <id,f,addr> that arrives at a switch there is a unique reply message <id,val> that eventually returns to the switch from memory. This is easy to prove by induction.

We assume that the arrival order required for individual messages is preserved for combined messages: if a_1 and a_2 are two successive request messages sent by a processor to the same memory location, and some node receives two combined messages, one representing a_1 and the second representing a_2, then the message representing a_1 arrives first. This condition is trivially satisfied for networks with unique paths.

Lemma 4.1. Consider a combining memory system. Let $A = <id,addr,f>$ be a memory request message, representing requests $a_1 = <id_1,addr,f_1>, \cdots, a_n = <id_n,addr,f_n>$. Let a_i' be the reply message associated with a_i, i.e. the reply message $<id_i,val>$ received by the processor that issued a_i. Then

(1) $f = f_1 o \cdots o f_n$;

(2) The values returned by all of the a_i' are the same as would be returned if the memory accesses associated with requests $a_1, \cdots, a_n$ were executed consecutively in memory.

(3) If request A reaches memory without being combined, the value stored at location addr after execution of request A is the same as the final value stored at location addr after consecutively executing the memory accesses associated with $a_1, \cdots, a_n$.

Proof: The lemma is proven by induction on the number of requests represented by a memory access message. It is trivial for a message that represents one request. Next, assume that the lemma is true for messages representing less than n requests, and assume that A is obtained by combining B and C, where B represents r requests and C represents $n-r$ requests $(1 \le r < n)$. Let $B = <id^B,addr,g>$ and $C = <id^C,addr,h>$, so that $A = <id^B,addr,goh>$. Then message A generates a reply $<id^B,val>$, which will also be the reply to request B; request C generates the reply $<id^C,g(val)>$. If A reaches memory then $val = @addr$ and the new value in memory is $h(g(@addr))$.

Let $b_1 = <id_1^b,addr,g_1>, \cdots, b_r = <id_r^b,addr,g_r>$ be the sequence of requests B represents; similarly, let $c_1 = <id_1^c,addr,h_1>, \cdots, c_{n-r} = <id_{n-r}^c,addr,h_{n-r}>$ be the sequence of requests C represents. Let b_i' and c_i' be the reply messages associated with the respective requests. By the inductive assertion, $g = g_1 o \cdots o g_r$ and $h = h_1 o \cdots o h_{n-r}$; the messages b_i' return the values val, $g_1(val)$, ..., $g_{r-1}(\cdots(g_1(val))\cdots)$; the messages c_i' return the values $g(val)$, $h_1(g(val))$, ..., $h_{n-r-1}(\cdots(h_1(g(val)))\cdots)$. It follows that the values returned, and the new memory value when A reaches memory are as if the memory accesses associated with $b_1, \cdots, b_r, c_1, \cdots, c_{n-r}$ were successively executed in this order. This proves the lemma. □

Theorem 4.2. A combining multiprocessor memory system is correct.

Proof: The previous lemma clearly implies the theorem: Indeed, let $M_1, \cdots, M_m$ be the successive memory request messages that are processed at a particular memory location. Replace each such message by the sequence of memory requests it represents. Let $R_1, \cdots, R_n$ be the resulting sequence. Let R and R′ be successive requests sent by the same processor. If these requests were not combined (i.e. are not represented by a unique message M_k), then the message representing R reached memory before the message representing R'; it follows that R occurs before R' in the sequence $R_1, \cdots, R_n$. If R and R' are represented by a unique message M_k, then a message representing R was combined at some switch with a message representing R', to obtain a new message M. The message representing R arrived first at that switch; hence R occurs before R' in the sequence of request represented by M. It follows that R occurs before R' in the sequence $R_1, \cdots, R_n$.

The final value of the memory location is as if the requests $R_1, \cdots, R_n$ were processed in that order; for each request R_i there is some reply message A_i that eventually returns to the processor (R_i and A_i have the same id); and the value of A_i is as the value that would obtain from processing request R_i after requests $R_1, \cdots, R_{i-1}$ were processed. □

Note that successive requests sent by a processor to the same memory location may be combined; the result is still correct.

5. APPLICATIONS

Suppose one intends to combine RMW requests with mappings from some family Φ of transformations. Composition can generate any mapping in the semigroup[4] $\overline{\Phi}$ spanned by[5] Φ. We need to have an encoding for the mappings in $\overline{\Phi}$ in which

(1) the computer representations of mappings from $\overline{\Phi}$ have reasonable size;

(2) the encoding of $f \circ g$ can be easily computed from the encoding of *f* and the encoding of g; and

(3) $f(a)$ can be easily computed from the computer representations of *f* and a.

We say that Φ is *tractable* if it fulfills these conditions.

We can formalize the notion of ‘‘tractable’’: Assume memory accesses involve memory words of fixed size w; then $\overline{\Phi}$ is a semigroup of mappings in 2^w, for each w. The semigroup Φ is ‘‘tractable’’ if there is an encoding of $\overline{\Phi}$ into bits, $\phi : \overline{\Phi} \rightarrow \{0,1\}^*$, such that

(1) $|\phi(f)| = O(w)$ (the encoding of a mapping requires a constant number of words);

(2) The computation of $\phi(f \circ g)$ from $\phi(f)$ and $\phi(g)$, and the computation of $f(a)$ from $\phi(f)$ and a are in the class NC; i.e. they can be computed by circuits of small size and depth, where small means size $w^{O(1)}$ and depth $\log^{O(1)} w$.

These conditions are trivially satisfied in all cases we consider.

5.1. Loads, Stores, and Swaps

We now show how to combine loads, stores, and swaps. Initially, we only show how to combine loads and swaps, since a store is simply a swap where the value returned is ignored. Recall that a load

---

[4] A semigroup is a set closed under an associative operation, which in this case is map composition.

[5] The set $\overline{\Phi}$ spanned by Φ is the smallest set containing Φ closed under composition.

from variable X is equivalent to RMW(X,**id**), where **id** is the identity mapping, and a swap of value local variable Y with shared variable X is equivalent to Y $\leftarrow$ RMW(X, $\mathbf{I}_Y$), where $\mathbf{I}_Y$ is the mapping that has constant value Y. The set of mappings $\{\mathbf{I}_v\} \cup \{\mathbf{id}\}$ is a semigroup, and composition is easily computed. A mapping from this semigroup is represented by one computer word and one opcode bit. The composition yields the expected results:

- A load followed by a load combine into a load.
- A load followed by a swap combine into a swap (the value fetched is returned to the load).
- A swap followed by a load combine into a swap (the value being stored is returned to the load).
- A swap followed by a swap combine into a swap of the second value.

One need not transmit the value returned by a store request, as it is of no interest; an acknowledgment suffices. One can avoid returning values by distinguishing between stores and swaps. Then, with the possible exception of an extra tag bit, combining never generates extra traffic; often it will decrease it significantly. The following 3×3 table shows how to combine loads, stores, and swaps.

	load	store	swap
load	load	swap	swap
store	store	store	store
swap	swap	swap	swap

Note that, in general, the order of combined requests is arbitrary and can be reversed. This can be used to decrease network traffic further. For example, if the network always chooses to effect a store before a load whenever two such requests are combined, then the store never needs to return a value. The same optimization applies when combining stores and swaps. The following 3×3 table shows how to combine loads, stores, and swaps, if the order of requests can be reversed. An * means the order of the operations is reversed.

	load	store	swap
load	load	store*	swap
store	store	store	store
swap	swap	store*	swap

Note, however, that this reversing operations is clearly wrong when successive requests of the same processor are combined.

The situation of a store combined with a load, suggests another slight improvement in performance: satisfy the load immediately. That is, the store would be forwarded to the memory module and its value will also be immediately returned, back to the processors that issued the load. However, this optimization is incorrect; the load request may be satisfied before the store occcurred in memory, leading to incorrect results. Consider, for example the following computation.

Processor 1	*Processor 2*	*Processor 3*
(1) $A \leftarrow 1$	(2) $a \leftarrow A$	(4) $b \leftarrow B+1$
	(3) $B \leftarrow a$	(5) $A \leftarrow b$

Assume that the memory works correctly, and that processors respect data dependencies. Assume that initially A=B=0. Then the execution of this code cannot end with b=2 and A=1: the memory access in (2) must occur before the access in (3), and the access in (4) must occur before the access in (5); if b=2 then the memory accesses to A and B occurrred in the order 12345, and the last store done by instruction (5) assigns the value 2 to A. On the other hand, if the suggested optimization occurs, it may happen that memory is accessed in the order 23451, but nevertheless the load in (2) returns the value 1 stored by (1). In such a case we end with b=2 and A=1. Note that the incorrect execution may occur even if processors do not overlap successive memory accesses.

It is always possible to add load, store, and swap operations to a family of RMW operations, and combine them all, without greatly increasing the complexity of the system. More formally, if Φ is a semigroup of mappings, then $\Phi \cup \{\mathbf{I}_v\} \cup \{\mathbf{id}\}$ is a semigroup too. We have

$$f \circ \mathbf{id} = \mathbf{id} \circ f = f \ ,$$

$$f \circ \mathbf{I}_v = \mathbf{I}_v \ , \quad \text{and}$$

$$\mathbf{I}_v \circ f = \mathbf{I}_{f(v)} \ .$$

Thus, if Φ is tractable, then $\Phi \cup \{\mathbf{I}_v\} \cup \{\mathbf{id}\}$ is tractable.

Our discussion has assumed that loads, stores, and swaps always affect an entire memory cell (word of memory). If we assume a word-addressable machine, say with four byte words, then combination of store operations that affect only bytes or half-words will require introducing store operations that affect any subset of bytes in a word. At a higher level, if one combines atomic stores that affect components of a structured variable then one needs to support stores that affect an arbitrary subset of the components of this variable.

5.2. Associative Operations

Let θ be an associative operation. Then *fetch-and*-$\theta(X,a)$ is equivalent to RMW(X,θ_a), where $\theta_a(x) = x\ \theta\ a$. The function *fetch-and*-$\theta(X,a)$ corresponds to the indivisible execution of the following code.

```
function fetch-and-θ(X,a)
  begin
    temp ← X;
    X ← Xθa;
    return(temp)
  end
```

A fetch-and-$\theta(X,a)$ followed by fetch-and-$\theta(X,b)$ can combine into fetch-and-$\theta(X,a\theta b)$, since

$$\theta_a \circ \theta_b(x) = \theta_b(\theta_a(x))$$
$$= (x\ \theta\ a)\ \theta\ b$$
$$= x\ \theta\ (a\ \theta\ b) \qquad \text{(since } \theta \text{ is associative)}$$
$$= \theta_{a\theta b}(x)\ .$$

Thus, the semigroup $\{\theta_a\}$ is tractable whenever θ can be computed by a small circuit.

Perhaps the most important fetch-and-θ primitive for large-scale shared memory machines is the fetch-and-add, which was discussed earlier. The mapping can be represented by one computer word (the addend). Two other potentially useful fetch-and-θ primitives are *fetch-and-OR*, where *OR* is Boolean addition, and *fetch-and-min*. Fetch-and-OR(X,1) is the test-and-set operation. Fetch-and-min is useful for allocation with priorities.

5.3. Boolean Operations

The sixteen Boolean operations can also be combined, despite the fact that some of them are not even associative operations. Moreover, each of the operations can be applied to bit vectors, of one word size. We will first consider the unary Boolean operations.

Let Φ be the set of four Boolean functions on one variable, $\mathbf{0}$, $\mathbf{1}$, $\mathbf{x}$, and $\bar{\mathbf{x}}$. The associated RMW operations are *test-and-clear*, *test-and-set*, *load*, and *test-and-complement.* The four functions in Φ can be represented by two bits, and can be composed using the following 4×4 table.

	load	clear	set	comp
load	load	clear	set	comp
clear	clear	clear	set	set
set	set	clear	set	clear
comp	comp	clear	set	load

The function compositions can be computed in hardware with few gates. Thus Φ is a tractable semigroup.

As a result all sixteen operations fetch-and-θ, where θ is a binary Boolean operation, can be combined. The reason is that the value of the second variable is fixed to a constant (0 or 1) when a request is issued, and every Boolean operation on two variables with one of the variables fixed is equivalent to some Boolean operation on one variable. For example, fetch-and-AND(X,a) is a load when $a = 1$, and is a test-and-clear when $a = 0$.

This result can be extended to Boolean operations on bit vectors. Mappings on bit vectors of length n are represented by $2n$ bits. Such operations are useful to support multiple locking.

5.4. Arithmetic Operations

Let Ψ be the set of arithmetic operations addition, subtraction, multiplication, and division. We also put into Ψ the reverses of the two noncommutative operations: reverse subtraction of a and b is $b-a$, and reverse division of a and b is b/a. We wish to support and combine all the operations of the form *fetch-and-*ψ, where $\psi \in \Psi$. In order to do that, we need to support and combine the operations RMW(X, ψ_a), where $\psi \in \Psi$, and $\psi_a(X) = X \psi a$. The semigroup spanned by the set of mappings $\{\psi_a : \psi \in \Psi\}$ consists of the Moebius functions: These are the functions of the form

$$x \to \frac{ax+b}{cx+d} \ ,$$

where a, b, c, and d are constants, and either $c \neq 0$ or $d \neq 0$.

We represent the function

$$x \to \frac{ax+b}{cx+d}$$

by the 2x2 matrix of coefficients

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} .$$

If f_A is the Moebius function represented by the matrix A, then

$$f_B \circ f_A = f_{AB} .$$

Thus, a function is represented by four coefficients, and two functions are composed by multiplying two 2x2 matrices.

We can now efficiently support all assignments of the form $x \leftarrow x \theta c$, where θ is an arbitrary arithmetic operation, and c is a constant or a private variable. These assignments will be executed atomically, while still being combined in the network. Such assignments form a large part of the machine code in typical applications.

If one wishes to support only addition and multiplication, then it is sufficient to consider functions of the form

$$x \to ax+b \ ,$$

which can be represented using only two coefficients. Combining two such mappings requires two multiplications and one addition.

Hardware arithmetic operations are not associative. Use of the associativity law may change occurrences of overflows in integer arithmetic, and may change occurrences of overflows, underflows, and rounding errors in floating point arithmetic. As our combining mechanism relies on associativity, the arithmetic might not produce the same results as would the serial order of the operations. Furthermore, the transformations used are not numerically stable when division occurs; they are numerically stable when divisions are left out. In that respect, our combining mechanism suffers from the same shortcomings as compiler optimization techniques that use transformations based on algebraic identities.

It is possible to obtain an accurate combining mechanism for fixed point operations, not including division, by adding one extra bit to the intermediate values, thereby increasing the range by a factor of two. If an overflow occurs in that increased range then an overflow would have occured in the serial execution of the operations in the restricted range. A similar technique of using guard bits will keep rounding errors under control when floating point operations not involving division are combined.

5.5. Full-Empty Bits

Accesses to shared variables can be synchronized using memory tags. For example, the HEP computer uses a *full-empty* bit at each shared memory word [Sm]. These bits can be used to synchronize accesses in a producer consumer fashion. Writing may be conditional on the location being empty; a successful write sets the (full-empty) bit. Reading may be conditional on the location being full; a successful read may clear the (full-empty) bit.

A load operation has the same effect in memory as the corresponding conditional load operation. We may therefore assume that load operations are always executed unconditionally: a processor can check the value of the full-empty bit returned by the load operation to determine if it was successful. A conditional store operation that fails returns a negative acknowledgement; the processor may resend it later.

In order to implement this synchronization mechanism, consider the four memory access operations (which are defined formally below) that form the basis of those in tagged memory architectures: load, load-and-clear, store-and-set, and store-if-clear-and-set.

Let the pair $(X, flag)$ represent the variable X and its associated full-empty bit $flag$. Temporarily assume that stores are actually implemented as swaps, i.e. they return the old value. In order to implement the operation set as RMW operations, one needs four types of mappings.

(1) The identity mapping for *load*: $(X, flag) \rightarrow (X, flag)$.

(2) The mapping for *load-and-clear*: $(X, flag) \rightarrow (X, 0)$.

(3) The mapping for *store-and-set*: $(X, flag) \rightarrow (v, 1)$.

(4) The mapping for *store-if-clear-and-set*:

$$(X, flag) \rightarrow \begin{cases} (v,1) & \text{if } flag = 0 \\ (X,1) & \text{if } flag = 1 \end{cases}$$

To close this set of mappings under composition, two more mappings must be included:

(5) The mapping $(X, flag) \rightarrow (v, 0)$ is a *store-and-clear*. It implements a store-and-set followed by a load-and-clear.

(6) The mapping

$$(X, flag) \rightarrow \begin{cases} (v,0) & \text{if } flag = 0 \\ (X,0) & \text{if } flag = 1 \end{cases}$$

is a *store-if-clear-and-clear*. It implements a store-if-clear-and-set followed by a load-and-clear.

These requests can now be combined. The combining logic is simple. Each of the six types of operations can be encoded by a short opcode, an address, and optionally a data word.

A store request carries one data value. A reply to a request needs to carry a data value only if the request is a load or a combined store that contains a simple load operation. If these store operations are handled specially, then the number of data values transmitted through a combining network will never exceed the number that would have been transmitted in an uncombining network.

There is a problem if the operation set includes a standard store operation, i.e. one that does not change the full-empty bit. If a store followed by a store-if-clear-and-set are to combine, it cannot be determined *a priori* whether the conditional store will actually succeed. One solution is to forward both store values. A better solution is simply to reverse the order of the requests (to be the store-if-clear-and-set followed by the store). These can be forwarded as a store-and-set operation.

Reversing the order does not always help. For example, if the operations store-if-clear and store-if-set are combined, both store values have to be forwarded. As we will see in the next section in a much more general context, even if we include all types of full-empty operations, no request will ever have to carry more than two store values.

We assumed in this section a *busy-waiting* model for synchronization: an operation that fails returns a negative acknowledgement; the processor may retry later. An alternative mechanism is to *queue* a request at memory until it is executable. This decreases the network traffic. However, unless some time-out mechanism is available at the memory controller, the hardware may deadlock.

Assume the two operations load-and-clear-if-set and store-and-set-if-clear are used to access memory in a queueing system. Memory accesses at a location are executed in a sequence of alternating loads and stores. Thus, a set of i load and j stores can be combined into $|i-j|+1$ operations: stores are combined with loads, with the excess of loads or stores staying uncombined. While combining is not guaranteed to reduce traffic in the worst case, one can expect it will do so in the average case.

5.6. Data-Level Synchronization

One can have more than two possible states (full and empty), and operations other than read and write on data. In a general *data-level synchronization scheme*, we have a semigroup Φ of mappings representing the RMW operations that can be executed, and a set S of states. Each variable is tagged by its state. The execution of an operation on a variable is conditional on its being in a suitable state; the operation also changes the variable's state.

This mechanism can be represented by an automaton $A = <\Phi,S,\delta>$, where $\delta:S\times\Phi \rightarrow S$ is the state transition function. Assume that variable X is in state s, and an RMW(X,f) operation is issued. If $\delta(s,f) = \varepsilon$ (i.e. undefined) the operation fails, and a negative acknowledgement is returned. Otherwise, RMW(X,f) is executed, and the new state of X is set to $\delta(s,f)$. Define the mapping f' by

$$f'(X,s) = \begin{cases} (f(X), \delta(s,f)) & \text{if } \delta(s,f) \neq \varepsilon \\ (X, s) & \text{otherwise} \end{cases}$$

Then the execution of the operation RMW(X,f) under the control of the automaton **A** is equivalent to the execution of the operation RMW($(X,s),f'$).

Consider now the case where the operations executed are stores and loads. The basic operations are then

- load(X,V,δ): Load from X if state s is in V and change state to $\delta(s)$.
- store(X,v,V,δ): Store the value v into X if state s is in V and change state to $\delta(s)$.

For uniformity, we represent a load by the tuple (X,Ω,V,δ), where the special value Ω represents the fact that no store is executed. A combined request then has the form <X, (v_1,V_1,δ_1), ..., (v_k,V_k,δ_k)>, where the V_i are disjoint sets of states. The meaning of this operation is: if state $s \in V_i$ and $s \notin S_1, \cdots, S_{i-1}$ then store v_i (or store nothing if $v_i = \Omega$) and change to state $\delta_i(s)$. If s is not in any V_i, then the operation fails.

A combined operation that represents k atomic store operations carries at most k store values. Also, a combined operation never carries more than $|S|$ store values, where $|S|$ is the number of states of the controlling automaton A. This is in general the best possible bound: if there is an operation *store-if-state*$=s$ for each state s of A, then a combined store may have to carry a distinct store value for each state. This is tractable when the number of states is small, such as when a full-empty bit is used; it is not tractable when the number of states is large. For example, the synchronization primitives defined by Zhu and Yew [ZY] for the Cedar machine at the University of Illinois and by Pier and Gajski [GP] use full word tags. With m bit tags, there are 2^m possible states, and 2^m is the best possible uniform bound on the number of store values in a combined request.

Memory accesses controlled by a regular automaton can be used to support *simple path expressions* [CH]. Path expressions are used to synchronize access to shared objects. For each such object there is a set of possible operations on it. A regular expression over the alphabet consisting of these operations defines the language of legal sequences of operation applications on each object.

A deterministic automaton corresponding to the path expression is built. Each object is represented by a variable in memory, to which access is protected by this automaton. Each execution of a protected operation is preceded by an access to that variable that performs the corresponding automaton transition. Then the executions of the operations are sequenced according to the path expression. The mechanism suggested in this section allows an efficient implementation of such a system.

6. RMW AND PARALLEL PREFIX

This section shows the relationship between the combining mechanism presented in this paper with a well known computational problem, prefix computation. The combining logic turns out to be an asynchronous version of a well known parallel synchronous algorithm. This sheds further light on performance aspects of combining.

Consider successive execution of the operations RMW(X,f_1), ..., RMW(X,f_n). These operations return the values X, $f_1(X)$, ..., $f_{n-1}(\cdots(f_1(X))\cdots)$; the value $f_n(\cdots(f_1(X))\cdots)$ is stored in memory. Thus, execution of these operations amounts to the computation of X, $f_1(X)$, ..., $f_n(\cdots(f_1(X))\cdots)$ or, equivalently, to the computation of $\mathbf{I}_X$, $\mathbf{I}_X \circ f_1$, ..., $\mathbf{I}_X \circ f_1 \circ \cdots \circ f_n$. This is a particular instance of the *prefix computation* problem [LaF]: given x_1, ..., x_n, compute x_1, $x_1 * x_2$, ..., $x_1 * \cdots * x_n$, where the operation * is an arbitrary associative operation. In our case, * is map composition.

Prefix computation when solved in parallel is known as *parallel prefix*. The memory access mechanism proposed in this paper provides in fact a parallel solution to the prefix computation problem. The computations are performed on the nodes of a tree in the interconnection network that connects the processors to one memory module. In a multistage network, in which processors have at most one outstanding request to each memory location, this is a physical tree, which is a subgraph of the network. In other cases this is a virtual tree which is embedded in the interconnection network graph.

The problem solved by the combining network differs from parallel prefix in that the order of the elements combined (with the exception of the first) is arbitrary. By ordering the operations correctly, one obtains a distributed, asynchronous network that solves the parallel prefix problem.

The computation is performed on a network of processes connected as a (not necessarily complete) binary tree with n leaves. The inputs are stored at the n leaves of a binary tree, which corresponds to the processors of the parallel computer. The root of the tree has one parent, called *superoot*; it corresponds to the memory module that contains the variable accessed; the internal nodes of the tree correspond to the combining switches in the processor to memory interconnection network. We describe below in CSP notation [Ho] the different types of processes.

Leaf Process

```
[Leaf:: val;
    parent ! val;
    parent ? val
]
```

Internal Node Process

```
[Node:: lval, rval, pval;
    left_child ? lval;
    right_child ? rval;
    parent ! lval*rval;
    parent ? pval;
    left_child ! pval;
    right_child ! pval*lval
]
```

Superoot Process

```
[Superoot:: val;
    child ? val;
    child ! id
]
```

Let val_i be the initial value at the *i*-th leaf. At the end of the computation the value at the *i*-th leaf equals to $\mathrm{val}_1 * \cdots * \mathrm{val}_{i-1}$; the value at the superoot equals to $\mathrm{val}_1 * \cdots * \mathrm{val}_n$.

If the tree is complete, then the operations performed by this tree are exactly the same operations performed by the Ladner-Fisher parallel prefix network [LaF]. The global clock synchronization used by their algorithm is replaced by local data-flow synchronization. Each internal node performs two multiplications, of which $\lceil \lg n \rceil$ are trivial. Thus, $2n - 2 - \lceil \lg n \rceil$ nontrivial multiplications are done. The algorithm can be implemented to run in $2 \lceil \lg n \rceil - 2$ multiplication cycles, when globally synchronized.

7. CONCLUSION

This paper provides a formal method for reasoning about the correctness of parallel computer architectures. It applies it to a concrete problem, arising in the design of machines such as the NYU Ultracomputer and the IBM RP3. Formal tools have been widely used to reason about concurrent software and communication protocols. On the other hand, there are few, if any, applications to parallel computer architectures. We hope our work will encourage a more rigorous approach to the definition of parallel architectures.

The paper provides the theoretical underpinnings of the combining mechanism used by the NYU Ultracomputer and RP3. It presents a general formulation of RMW operations and of combining for these operations. On the other hand, we have not discussed implementations of the combining logic. An implementation of an efficient switch that supports combining of fetch-and-add requests is described in [DKS], [DKSS]. This design has been realized in custom VLSI at NYU. The same scheme can be used for other RMW operations. This realization, while reasonably fast, requires significant amounts of supplementary hardware (over that required for packet switching). Note that one can use combining logic that detects only part of the combinable pairs. Memory accesses are correctly performed even with partial combining, or no combining at all. Thus, different cost-performance tradeoffs are possible.

Combining or partial combining can be used on a wide variety of interconnection networks. The only major restriction is that requests must return via the same route (although in the reverse direction). Thus, the mechanisms described in this paper can be easily adopted for use by direct connection machines, such as the cosmic cube [Se], where the processors themselves act like network switches and the local memories at each node are all view as part of a distributed, shared memory. Combining can also be used on machines where multiple processors are connected to a shared memory by a bus. The shared memory is often heavily interleaved; thus, it achieves high, but uneven throughput. A FIFO buffer is often used to decouple memory from the shared bus. Combining is this queue will improve the memory throughput by reducing conflicting accesses to the same memory bank.

REFERENCES

[CH] Campbell, R. H. and A. N. Haberman, The Specification of Process Synchronization by Path Expressions, *Lecture Notes in Computer Science, vol. 16,* Springer-Verlag, pp. 89-102, Aug. 1976.

[Co] W. Collier, Principles of Architecture for Systems of Parallel Processes, IBM Tech. Rep. TR00.3100, March 1981.

[DKS] Dickey, S., R. Kenner, and M. Snir, An Implementation of a Combining Network for the NYU Ultracomputer, Ultracomputer Note #93, NYU, Jan. 1986.

[DKSS] Dickey, S., R. Kenner, M. Snir, and J. Solworth, A VLSI Combining Network for the NYU Ultracomputer, *IEEE Proc. of the Intl. Conf. on Computer Design,* pp. 110-113, Oct. 1985.

[Di] Dijkstra, E. W., Hierarchical ordering of sequential processes, *Acta Informatica, vol. 1,* pp. 115-138, 1971.

[DGSS] Draughon, E., R. Grishman, J. Schwartz, and A. Stein, Programming Considerations for Parallel Computers, Rep. IMM 362, Courant Inst. of Mathematical Sciences, New York Univ., New York, N. Y., 1967.

[GP] Gajski, D. D. and J.-K. Peir, Essential Issues in Multiprocessor Systems, *IEEE Computer,* pp. 9-28, June 1985.

[GGK] Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, The NYU Ultracomputer -- Designing an MIMD Parallel Computer, *IEEE Transactions on Computers, vol. C-32,* pp. 75-89, 1984.

[GK] Gottlieb, A. and C. P. Kruskal, Coordinating Parallel Processors: A Partial Unification, *SIGARCH,* Oct. 1981.

[GLR] Gottlieb, A., B. D. Lubachevsky, and L. Rudolph, Efficient Techniques for Coordinating Sequential Processors, *ACM TOPLAS,* pp. 164-189, 1983.

[Ho] Hoare, C. A. R., Communicating Sequential Processes, *CACM vol. 21,* pp. 666-677, Aug. 1978.

[La1] Lamport, L., How To Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *IEEE Transactions on Computers, vol. C-28,* pp. 690-691, 1979.

[La2] Lamport, L., Time, Clocks, and the Ordering of Events in a Distributed System. *CACM,* pp. 558-565, July 1978.

[La3] Lamport, L., On Interprocess Communication, to appear in *Distributed Computing.*

[LaF] Ladner, R. and M. J. Fisher, Parallel Prefix Computations, *JACM vol. 27,* pp. 831-838, 1980.

[LKK] Lee, G., C. P. Kruskal, and D. J. Kuck, The Effectiveness of Combining in Multistage Interconnection Networks in the Presence of ‘Hot Spots’, *1986 International Conference on Parallel Processing,* pp. 35-41, Aug. 1986.

[LyF] Lynch, N. and M. J. Fisher, On Describing the Behavior and Implementation of Distributed Systems, *Theoretical Computer Science vol. 13,* pp. 17-43, 1981.

[MM] Milne, G. and R. Milner, Concurrent Processes and their Syntax, *JACM vol. 26,* pp. 764-772, 1979.

[PG] Peir, J.-K. and Gajski D. D., Data Flow Execution of Fortran Loops, *Proc. 1st Intl. Conf. on Supercomputing Systems,* Dec. 16-20, 1985.

[PET] Peterson, J. and A. Silbershatz, *Operating System Concepts*, Addison-Wesley, 1983.

[PBH] Pfister, G. H., et al., The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, 1985 Intl. Conf. on Parallel Processing, pp. 764-772.

[PN] Pfister, G. H., and A. Norton, 'Hot Spot' Contention and Combining in Multistage Interconnection Networks, IEEE Transactions on Computers vol. C-34, pp. 933-938, 1985.

[RT] Rettberg, R. and R. Thomas, Contention is no Obstacle to Shared-Memory Multiprocessing, Com. ACM 29, 12, pp. 1202-1212, 1986.

[Ru] Rudolph L., Software Structures for Ultraparallel Computing, PhD Thesis, New York University, 1981.

[Se] Seitz, C., The Cosmic Cube, *CACM vol. 28* pp. 22-33, Jan. 1985.

[SS] D. Shasha and M. Snir, Efficient and Correct Execution of Programs That Share Memory, Ultracomputer Tech. Rep., NYU, Feb. 1986.

[Sm] Smith, B. J., Architectures and Applications of the HEP Multiprocessor Computer System, *Real-Time Signal Processing IV, SPIE,* pp. 241-248, 1981.

[SBK] Sullivan, H., T. R. Bashkow, and D. Klappholz, A Large Scale, Homogeneous, Fully Distributed Parallel Machine, *The 4th Ann. Symp. on Computer Architecture*, pp. 105-134, 1977.

[ZY] Zhu and Yew, A Synchronization Scheme and Its Applications for Large Scale Multiprocessor Systems, *Proc. Conf. on Distributed Computing Systems,* pp. 486-491, 1984.