PARALLEL OPERATIONS

Paul F. Reynolds, Jr.
Craig Williams
Raymond R. Wagner, Jr.

Computer Science Report No. TR-89-16
December, 1989

Parallel Operations*

Paul F. Reynolds, Jr.
Craig Williams
Raymond R. Wagner, Jr.

ABSTRACT

We define a new synchronization primitive for parallel asynchronous computations, the parallel operation. The parallel operation (parop) is a special case of the atomic action. A parop is a set of partially ordered independent operations that appears to be executed indivisibly. Depending on the programming model, each operation is either a message to a process or an access to a shared variable. Operations must be independent in the sense that the process issuing a parop must be able to issue all the operations in the parop before any operation in that parop is executed. The operations in a parop appear to be executed in a single indivisible step even though different operations in the parop may be executed at different nodes. We describe a mechanism, local synchrony, that supports a deadlock-free, fair, and highly concurrent implementation of parops in hardware. Using this mechanism, parops can be implemented without locking. Parops can also be pipelined. Local synchrony supports the sequentially consistent execution of pipelined memory accesses. It is compatible with combining and can be adapted to a wide class of network topologies. Since it requires no global clock or other global arbitration, local synchrony is suitable for implementing parops on distributed as well as tightly coupled systems. The use of parops reduces the need for locks in accessing shared memory or distributed objects. The parop also forms a basis for a more general method for coordinating access to shared memory described in a companion paper.

1. INTRODUCTION

Atomic access to multiple data objects is a powerful programming paradigm. Parallel assignments, transactions, critical regions, and monitors are a few of the methods devised to support atomic actions in an asynchronous parallel computation. This paper defines a new synchronization primitive, the parallel operation (parop), for specifying a class of atomic actions and proposes a mechanism, local synchrony, for implementing parops in hardware without locking.

The parop is a special case of the atomic action, a set of instructions executed as an indivisible step. The atomic action is the smallest unit of process interleaving [OwL82]. No process other than the process executing the atomic action can observe or change an intermediate state in the execution of an atomic action. Our claim that parops are executed atomically rests on an assumption of hardware reliability. We do not address the problem of recovery from hardware faults.

* This work is supported in part by a grant from the Virginia Engineering Foundation and by an assistantship in parallel processing sponsored by DARPA and NASA and administered by the University of Maryland.

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. Note that parops can not specify atomic execution of the statement `A := B`, where `A` and `B` are shared variables. Execution of this statement requires two operations, a read to `B` and a write to `A`, but the operations cannot be issued in the same parop because the write cannot be issued until after the read is executed.

In spite of this limitation, we propose the parop as a synchronization primitive for the following reasons:

(1) Parops are powerful in combination with other mechanisms. In a companion paper [WiR89], we show how to use parops in combination with split operations and access sequences to support process synchronization and a much broader class of atomic actions.

(2) Parops can be implemented efficiently in hardware. No locking of accessed objects is required and multiple processes can execute them concurrently.

We describe the parop in this paper from the viewpoint of a shared memory computation on a tightly coupled multiprocessor, but it can be adapted to other architectures and programming models.

- Parops are applicable to both the shared memory model (SMM) and message based model (MBM) of computation. In this paper we assume that each operation in a parop is an instruction accessing a shared variable, but parops can be adapted to the MBM by defining operations to be messages.
- Parops are applicable to distributed as well as to tightly coupled systems. Since our proposed implementation does not require a global clock or other global arbitration, parops are a reasonable synchronization primitive to propose for geographically distributed systems.
- Parops can be implemented on a wide class of static and dynamic network topologies, including dance hall topology multiprocessors, hypercubes, and meshes.
- Parops are compatible with combining networks such as the NYU Ultracomputer. The implementation of parops does not require a combining network, but on a network that supports combining,

operations from different parops that address the same shared variable can be combined, further increasing the concurrency of access to shared memory.

- Parops are scalable. Because parops are implemented without locking, they are especially well suited for use in massively parallel systems, where locking shared variables can seriously degrade performance.

The mechanism, local synchrony, with which we propose to implement parops is similar to that proposed by Awerbuch [Awe85] and Ranade [Ran87, RBJ88]. We use local synchrony as a way to incorporate some of the advantages of synchronous computation in an asynchronous computation. Processes can execute independently subject to data dependencies, but the routing of concurrently issued operations is loosely synchronized. We show that this synchronization supports not only a limited form of atomic action, but also sequentially consistent execution of pipelined operations.

The paper is organized as follows. Sections 2 and 3 define the syntax and semantics of parops. Section 4 describes a mechanism for implementing parops. Section 5 describes applications and concluding remarks follow in section 6.

2. NOTATION

Syntactically, a parop is a list of operation sequences that we will call macro-ops. The list is terminated by a semicolon and double bars "||" separate adjacent macro-ops within the list.

```
<parop> ::= <macro_op>
          | <parop> || <macro_op>;
```

Each macro-op is an ordered pair in which the first element names a shared variable and the second lists operations on that variable.

```
<macro_op> ::= <shared variable> : <operation_list>.
```

An operation list is a sequence of one or more operations on the same shared variable. Each operation specifies zero or more operands and adjacent operations are separated by commas.

Each operation specifies a step that is executed indivisibly. In this paper, we assume that each operation accesses a single location in shared memory. Accesses need not be limited to reads and writes, but can be any read-modify-write (RMW) operation. A RMW operation atomically reads and updates a single shared variable. The old value is returned and a new value assigned that is a function of the old value and any operands supplied with the operation [KRS88]. If the memory modules have sufficient computational power to compute the function, a RMW operation can be executed without locking. Note that reads and writes are special cases of the RMW operation.

We use the following notation to specify read and write accesses:

```
<operation> ::= read(<local variable>) | write(<expression>)
```

The operand to the `write` operation is the value to be written. The operand to the `read` is an address-valued expression identifying the local variable assigned the result of the read. For example, the macro-op `A:read(v)` assigns the value of shared variable `A` to local variable `v`. In this paper, identifiers for shared variables are in upper case and for local variables in lower case.

The location of variables and the value of operands can be dynamically computed using constants and local variables. This restriction ensures that operations are independently issuable.

3. SEMANTICS

A parop is a partially ordered set of operations that is executed atomically. The macro-ops within a parop are executed in any order, but operations within each macro-op are executed in the order in which they are listed. For example, the parop

```
A:read(v) || B:read(v),write(5);
```

nondeterministically assigns to the local variable `v` either the value of `A` or `B`, but if it assigns the value of `B`, the value assigned will be the ''old'' value of `B` before the assignment specified by the `write(5)` operation is executed. The shared variables named in a parop should be distinct. If two or more macro-ops in a parop name the same shared variable, the order in which the macro-ops are executed is unknown.

A parop is elaborated before any operation in the parop is issued, i.e., the first step in executing a parop is the evaluation of every expression in the parop representing the location of a variable or the value of an operand. For example, the macro-op `A[v]:read(v)` is equivalent to the assignment `v := A[v]`. The address of `A[v]` is computed before `v` is assigned the new value.

Each parop is executed atomically and in an order that is consistent with the order specified by the program. For example, consider the following parop program, where process P_1 issues parops $PO_{1,1}$ and $PO_{1,2}$, and P_2 issues $PO_{2,1}$ and $PO_{2,2}$:

```
P1 ::  A:write(1)  || B:write(1);                      PO1,1
       A:read(v1)  || C:read(v2),write(1);             PO1,2
P2 ::  C:read(v1);                                     PO2,1
       A:read(v2)  || B:read(v3);                      PO2,2
```

Assuming `A`, `B`, and `C` are 0 initially and that no other process accesses these variables, the following are true for every execution of the program:

- The parops appear to be executed indivisibly. In particular, execution of $PO_{1,1}$ and $PO_{2,2}$ cannot be observed to interleave. Process P_2's reads in $PO_{2,2}$ will return either 0 for both `A` and `B` or 1 for both `A` and `B`.

- Any pair of operations from the same macro-op appears to be executed in the order in which they appear textually. Note that $PO_{1,2}$ contains both read and write operations on `C`. Since the read precedes the write, the read is executed first and will return 0.

- Pairs of parops issued by the same process appear to be executed in the order in which they are issued. Thus the read to `A` in $PO_{1,2}$ will return the value 1 written by $PO_{1,1}$. Furthermore, if the read to `C` in $PO_{2,1}$ returns 1, indicating that $PO_{1,2}$ was executed before $PO_{2,1}$, then the reads to `A` and `B` in $PO_{2,2}$ will return 1, not 0. Similarly, if the reads to `A` and `B` in $PO_{2,2}$ return 0, the read to `C` in $PO_{2,1}$ will also return 0.

More formally, 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]. For every execution of a parop program, there is an equivalent execution of the same program in which parops are executed serially, i.e., without interleaving of operations from different parops, in an order that is consistent with the program and in which the operations within each macro-op are executed in the order in which they appear textually.

Definition. For any operation, OP_i, $parop(OP_i) = PO_j$, if OP_i is in parop PO_j.

Definition. Parop PO_i ''logically precedes'' PO_j, denoted $PO_i \rightarrow_p PO_j$, if the program specifies that PO_i be executed before PO_j, i.e., if PO_i and PO_j are issued by the same process and PO_i is issued before PO_j. We define ''logically precedes'' for operations analogously, but restrict the domain of the $\rightarrow_p$ relation to parops. For any pair of operations, OP_i and OP_j, OP_i ''logically precedes'' OP_j if $parop(OP_i) \rightarrow_p parop(OP_j)$ or if $parop(OP_i) = parop(OP_j)$ and OP_i textually precedes OP_j in the same macro-op.

Definition. For any pair of parops, PO_i and PO_j, let S be the intersection of the sets of shared variables named in PO_i and PO_j.

Definition. Parop PO_i ''is executed before'' PO_j, denoted $PO_i \rightarrow_e PO_j$ if
(1) for every variable v in S, S not empty, PO_i's operations on v are executed before PO_j's operations on v; or
(2) there exists a PO_k such that $PO_i \rightarrow_e PO_k$ and $PO_k \rightarrow_e PO_j$.

Definition. Parop PO_i ''precedes'' PO_j, denoted $PO_i \rightarrow PO_j$, if
(1) $PO_i \rightarrow_e PO_j \vee PO_i \rightarrow_p PO_j$; or
(2) there exists a PO_k such that $PO_i \rightarrow PO_k$ and $PO_k \rightarrow PO_j$.

In every execution of a parop program, the following three properties, the ''parop properties'' hold:

Property P1. For every pair of parops, PO_i and PO_j, S is not empty $\Rightarrow PO_i \rightarrow_e PO_j \vee PO_j \rightarrow_e PO_i$.

Property P2. The $\rightarrow$ relation is asymmetric, i.e., $PO_i \rightarrow PO_j \Rightarrow PO_j \not\rightarrow PO_i$, where $PO_j \not\rightarrow PO_i$ means $\neg(PO_j \rightarrow PO_i)$.

Property P3. For every pair of operations, OP_i and OP_j, that access the same variable, OP_i logically precedes $OP_j \Rightarrow OP_i$ is executed before OP_j.

Note that the $\rightarrow$ relation is transitive by definition and is asymmetric, and hence irreflexive, by **P2**. It is, therefore, a partial order.

Let E_p be any execution of a parop program and E_s be any execution of the same program in which the parops are executed serially in an order that extends the partial order $\rightarrow$ induced by E_p and in which operations from the same parop that access the same variable are executed in the order in which they are executed in E_p. Since E_s extends the partial order $\rightarrow$ induced by E_p, the order in which parops are executed in E_s is consistent with the order specified by the program.

We show that every execution of a parop program is sequentially consistent by showing that E_p is "access equivalent" to E_s. Two executions are access equivalent if they contain the same operations and if every pair of operations that access the same shared variable are ordered in the same way in both executions. Access equivalence is closely related to the concept of conflict equivalence in concurrency control for databases. Two schedules are conflict equivalent if they contain the same transactions and if every pair of conflicting steps is ordered in the same way in both schedules, where two steps conflict if they access the same entity and do not commute, e.g. a read and a write [Pap86]. We define access equivalence in the same way, but assume that every pair of accesses to the same shared variable conflict.

An execution of a parop program, E_p, is access equivalent to E_s because for any pair of operations, OP_i and OP_j, that access the same variable, OP_i and OP_j are executed in the same order in both E_p and E_s. Assume, without loss of generality, that OP_i is executed before OP_j. If the operations are from different parops then, by **P1**, $parop(OP_i) \rightarrow_e parop(OP_j)$. Since the order in which parops are executed in E_s extends the partial order $\rightarrow$ induced by E_p, OP_i is also executed before OP_j in E_s. If the operations are from the same parop, then by definition, E_s also executes OP_i before OP_j.

The access serializability of parops implies that each parop appears to be executed indivisibly. But note that $PO_i \rightarrow_e PO_j$ does not imply that all of the operations in PO_i are executed before any of the operations in PO_j. Consider the following two parops.

P_1:: `A:write(1) || B:write(1);` PO_1
P_2:: `A:write(2) || B:write(2);` PO_2

The semantics of parops allow these operations to be executed in the following interleaved order:

```
A:write(1), A:write(2), B:write(1), B:write(2)
```

Even though there is no instant in which `A = B = 1`, the serializability of parops ensures that any process that reads `A` and `B` in the same parop will get a consistent view. A similar phenomenon arises with some concurrency control protocols, e.g., the tree protocol [SiK80], where each transaction is ensured a consistent view of the database even though there may be no instant in which the database is in a consistent state.

4. IMPLEMENTATION

An implementation of parops must ensure that execution conforms with the three parop properties. Clearly parops can be implemented using locks and delays to ensure atomicity and sequential consistency. But if parops are to be a useful synchronization primitive, the implementation must be efficient. In this section we propose a mechanism, local synchrony, for implementing parops in hardware without locks or global arbitration.

Local synchrony is a technique for simulating a synchronized network on a network with no global clock. Each node in the network advances its local logical clock one tick by sending a control signal, a "token," to each of its neighbors. Each node synchronizes with its neighbors using the following rule: emit message M at logical time i only after receiving every messages that may affect M emitted by a neighbor at logical time $i-1$. In general, any message emitted by a neighbor at time $i-1$ may affect a message emitted at time i, but in special cases, a node may have enough information to emit a message at time i before receiving all messages emitted by its neighbors at time $i-1$. Local synchrony has been used by Awerbuch to support the execution of SIMD graph algorithms on an asynchronous network [Awe85] and by Ranade to implement efficient combining [Ran87, RBJ88].

In this section, we show how to use local synchrony to implement parops. We first describe a general method applicable to any network topology in which the length of all "routable" paths between any given PE,MM node pair is the same. A routable path is a path that is consistent with the routing rules applied by the switches. Some physical paths may not be routable paths for a given routing algorithm. We then propose a simplified implementation for the important special case in which the length of all routable paths is the same.

4.1. A General Method for Implementing Parops

We consider a network of interconnected nodes, where each node contains a switch, zero or more PE's, and zero or more MM's. All communications within a node and between nodes is over reliable unidirectional FIFO links with finite but unbounded delay and there is at least one routable path of links from each PE to each MM and from each MM to each PE. The following properties hold for each routable path from a given PE to a given MM:

(1) If there is more than one routable path, then the paths are the same length in links.

(2) Every routable path must go through at least one switch. In particular, every operation issued by a PE must go through the node's switch even if it accesses a variable located in a MM on the same node.

The paths in the reverse direction, from MM to PE, need not be similarly constrained.

It is convenient to assume that a switch interface unit (SIU) is associated with each PE. The SIU prepares parops for routing by the switch. In any given implementation of parops, the functions of the SIU may be performed instead by the PE or switch. Each SIU knows the distance to each MM.

Assume each of n PE's is assigned a unique ID in the range $0 .. n-1$. Several processes may share a PE, but for simplicity we assume processes do not migrate. We require that processes issue operations in the order specified by the program and that no process is interrupted while it is issuing a parop. Note that as a consequence of these assumptions operations arrive at the SIU from the PE in an order that is consistent with the order specified by each process's sequential program and operations from different parops are not interleaved.

Every communication is either an operation, a token, or a response. A ''response'' is a packet issued by a MM. The implementation of parops does not require any special treatment of responses. A token is a control signal that can be distinguished from operations and responses. During the computation, each switch receives tokens and operations on its input lines and emits tokens and operations on its output lines and each SIU emits tokens and operations on its output line. No other links internal to the node carry tokens. We say that an operation is received (emitted) by a switch (SIU) in pulse i if it is received (emitted) by the switch (SIU) between the $i-1st$ and ith tokens received (emitted) on the line on which the operation is received (emitted). Note that the token number need not be explicitly recorded in the token if each switch (SIU) remembers the number of tokens that are received and emitted on each of its input and output lines respectively, where the first token received (emitted) on each line is token 0.

We define a labeling scheme for parops that is a total ordering of the parops and that can be extended to a total ordering of the operations. We show that the parop properties hold for any execution consistent with the labeling scheme and then describe algorithms for the SIU's and switches that ensure that every execution is consistent with the labeling scheme.

Labeling scheme. Each parop is labeled by the pulse in which operations in the parop arrive at their destinations, the position of the ID of the issuing PE in a given permutation of the PE ID's, and the rank of the parop among the parops issued by the issuing PE.

DEFINITION. For any parop, PO_i, $label(PO_i)$ is a 3-tuple $(pulse(PO_i), f(pe(PO_i)), rank(PO_i))$, where
$pulse(PO_i) = j$ if each operation in PO_i is received at its destination node in the jth token pulse, and is otherwise undefined;
f is a permutation of the PE ID's;
$pe(PO_i) = j$ if PO_i is issued by PE_j; and
$rank(PO_i) = j$ if PO_i is the jth parop issued by the PE that issues PO_i.

DEFINITION. For any pair of parops, PO_i and PO_j, $PO_i <_p PO_j$ if $label(PO_i)$ lexicographically precedes $label(PO_j)$.

If operations are from different parops, the ordering of the parops as determined by the labeling scheme determines the ordering of the operations. If operations are from the same parop, their order is determined by the order in which they appear textually in the parop.

DEFINITION. For any pair of operations, OP_i and OP_j, $OP_i <_o OP_j$ if $parop(OP_i) <_p parop(OP_j)$ or if $parop(OP_i) = parop(OP_j)$ and OP_i logically precedes OP_j.

Note that if the *pulse* function is defined for every parop, the $<_p$ relation is a total ordering of the parops and $<_o$ is a total ordering of the operations.

An execution is "consistent" with the labeling scheme if the following properties hold:

PROPERTY **L1.** The *pulse* function is defined for every parop.

Property L2. For any pair of parops, PO_i and PO_j, $PO_i \rightarrow_p PO_j \Rightarrow pulse(PO_i) \leq pulse(PO_j)$.

Property L3. For any pair of operations, OP_i and OP_j, that access the same variable, $OP_i <_o OP_j \Rightarrow OP_i$ is executed before OP_j.

Let E be an execution that is consistent with the labeling scheme. We show that the parop properties hold for E.

P1. By definition of the $<_o$ relation, for any pair of parops, PO_i and PO_j, $PO_i <_p PO_j \Rightarrow op(PO_i) <_o op(PO_j)$, where $op(PO_k)$ is any operation in PO_k. If $op(PO_i)$ and $op(PO_j)$ access the same variable, by **L3**, $op(PO_i) <_o op(PO_j) \Rightarrow op(PO_i)$ is executed before $op(PO_j)$ in E. Thus S not empty $\wedge\ PO_i <_p PO_j \Rightarrow PO_i \rightarrow_e PO_j$. Since $<_p$ is a total ordering of the parops, either $PO_i <_p PO_j$ or $PO_j <_p PO_i$ and **P1** holds.

P2. We first show that $PO_i \rightarrow PO_j$ implies $PO_i <_p PO_j$. There are three cases.

Case 1. $PO_i \rightarrow_p PO_j$. Parop PO_i and PO_j are issued by the same process and PO_i is issued before PO_j. Thus $f(pe(PO_i)) = f(pe(PO_j))$, $rank(PO_i) < rank(PO_j)$, and, by **L2**, $pulse(PO_i) \leq pulse(PO_j)$, labeling constraints that imply $PO_i <_p PO_j$.

Case 2. $PO_i \rightarrow_e PO_j$ and S not empty. For any given v in S, and operations $op(PO_i)$ and $op(PO_j)$ that access v, $op(PO_i)$ is executed in E before $op(PO_j)$. Since the operations access the same variable and E is consistent with the labeling scheme, by **L3**, $op(PO_i) <_o op(PO_j)$. Given that the operations are from different parops, $PO_i <_p PO_j$.

Case 3. Otherwise, there is a chain of one or more parops connecting PO_i and PO_j, where for each pair of parops, PO_r and PO_s, that are adjacent in the chain, either $PO_r \rightarrow_p PO_s$ or S not empty and $PO_r \rightarrow_e PO_s$. A simple induction on the length of this chain shows $PO_i <_p PO_j$.

Finally, we show that the $\rightarrow$ partial order induced by E is asymmetric. $PO_i \rightarrow PO_j \rightarrow PO_i \Rightarrow PO_i <_p PO_j <_p PO_i$, a contradiction since $<_p$ is a total order.

P3. Recall that operation OP_i logically precedes operation OP_j if $parop(OP_i) \rightarrow_p parop(OP_j)$ or if the operations are from the same parop and OP_i textually precedes OP_j in the same macro-op. If $parop(OP_i) \rightarrow_p parop(OP_j)$ then $parop(OP_i) <_p parop(OP_j)$, as shown above. In either case, OP_i logically precedes $OP_j \Rightarrow OP_i <_o OP_j$. By **L3**, OP_i is executed before OP_j in E.

Algorithms. We describe algorithms for the SIU's and switches that ensure every execution is consistent with the labeling scheme. For each operation, OP_i, received by a switch, the switch must be able to compute $f(pe(parop(OP_i)))$. The most general way to satisfy this requirement is to label each operation with the ID of the issuing PE. Note that this is the only label required. The pulse and rank of each operation are implicit.

SIU Algorithm. After emitting token 0, each SIU continuously analyzes parops received from the associated PE and emits operations and tokens on the output line to the switch. Operations can be emitted in any pulse and in any order subject to the following constraints:

SIU-1. For any parop, PO_i, let $D(PO_i)$ be the maximum distance in links that will be traveled by any operation in PO_i, counting the link from the SIU to the switch, and $T(PO_i)$ be the first pulse in which any operation from PO_i is emitted. For any pair of parops, PO_i and PO_j, if PO_i is received by the SIU before PO_j, then $T(PO_j) \geq T(PO_i) + D(PO_i) - D(PO_j)$.

SIU-2. For any operation, OP_i, let d_i be the distance to operation OP_i's destination. Operation OP_i is emitted in pulse $T(parop(OP_i)) + D(parop(OP_i)) - d_i$.

SIU-3. Every pair of operations that access the same variable and are emitted in the same pulse are emitted by the SIU in the same order in which they are received.

We describe an SIU algorithm that incorporates these constraints and that preprocesses the operation emitted in each pulse to lay the foundation for an efficient switch algorithm. After emitting the operations emitted in pulse i, the SIU analyzes any parops newly issued by the associated PE, if any. With each new parop, PO_i, the SIU determines $T(PO_i)$, choosing the earliest pulse that conforms with **SIU–1**. For each operation in the parop, the SIU determines, using **SIU–2**, the pulse in which to emit the operation, adding the operation to the end of a list of other operations scheduled to be emitted in the same pulse. The SIU then sorts the list of operations scheduled to be emitted in the next pulse, using the address of the shared variable accessed as the key, and emits token i followed by the sorted operation list. The sorting step is done using a stable sort to preserve, for each pair of operations that access the same variable, the order in which the PE issued the operations.

Switch Algorithm. Each switch routes the operations it receives onto an appropriate output line, subject to the following constraints:

SW-1. All operations received in pulse i are emitted in pulse $i+1$.

SW-2. For any pair of operations, OP_i and OP_j, that access the same variable and are received by the switch in the same pulse

(a) If the operations are issued by the same PE, the operations are emitted on the same output line;

(b) If the operations are received on the same input and emitted on the same output line, the operations are emitted in the order received.

(c) If the operations are received on different input lines and emitted on the same output line, OP_i is emitted before OP_j if $f(pe(parop(OP_i))) < f(pe(parop(OP_j)))$.

Note that the input lines from SIU's, if any, "look" to the switch just like any other input lines. The switch does not distinguish operations arriving on a line from an SIU from operations arriving on other input lines. For simplicity, we assume that the output lines to the MM's, if any, also "look" to the switch just like any other output lines. Actually, there is a difference. The MM's ignore tokens, so it is unnecessary for a switch to emit any tokens on an output line to an MM.

Since we do not require that there be a unique path between every PE,MM pair, there may be two or more paths between a given PE,MM pair that join and then split again at a switch. In general, the switch can choose to route an operation on any output line allowed by the routing algorithm, but for operations that access the same variable in the same pulse and are issued by the same PE, SW–2(a) requires that the operations be emitted on the same line.

Assuming that the SIU's emit operations within each pulse in sorted order, operations will remain sorted if each switch merge sorts the streams of operations arriving on its input lines in each pulse. Note that if operations were not received in sorted order, to ensure conformity to SW–2(c), each switch would have to buffer all operations received during a pulse before emitting any of the operations. The assumption that operations arrive in sorted order allows the switches to use a technique similar to wormhole routing [DaS87]. If the leading bits for each operation encode the destination, a switch can select the operation that accesses the variable with the minimum address from among the operations ready on each input line and emit that operation without first buffering it. Each token acts of the end of stream marker, i.e. it has address ''infinity.'' Let OP_i be the operation, among all the operations ready on each input line of a given switch, which accesses the variable with the least address. A switch waits if no input line has a token or operation ready or if there is some input line, in_i, with no token or operation ready and the last operation received on in_i accessed a variable with an address less than the address of the variable accessed by operation OP_i.

Consistency. We show that any algorithm for implementing parops that conforms with these constraints is consistent with the labeling scheme.

L1. The *pulse* function is defined for each parop. By **SW–1** an operation, OP_i, emitted by a SIU in pulse t is received at the switch in the destination node in pulse $t + d_i - 1$. Since by **SIU–2** each operation in $parop(OP_i)$ is emitted by the SIU in pulse $T(parop(OP_i)) + D(parop(OP_i)) - d_i$, each operation in any given parop is received at its destination in the same pulse, pulse $T(parop(OP_i)) + D(parop(OP_i)) - 1$.

L2. For any pair of parops, PO_i and PO_j, $PO_i \rightarrow_p PO_j$ implies PO_i is received at the SIU before PO_j. SIU–1 ensures that the *pulse* function increases monotonically for parops received at any given SIU.

L3. For any pair of operations, OP_i and OP_j, that access the same variable, assume without loss of generality that $OP_i <_o OP_j$. Then OP_i is executed before OP_j. There are three cases:

Case 1. If $pulse(parop(OP_i)) < pulse(parop(OP_j))$, then, by definition of the *pulse* function, OP_i is received at the destination switch, the switch on the same node as the MM containing the variable the operations access, in an earlier pulse than OP_j. Since the destination switch sends the operations to the MM in the order received and the MM executes the operations in the order received, OP_i is executed before OP_j.

Case 2. Otherwise, if $f(pe(parop(OP_i))) < f(pe(parop(OP_i)))$, then let switch S_{ij} be the last switch at which the paths taken by OP_i and OP_j join, i.e., OP_i and OP_j are received at S_{ij} in the same pulse on different lines and are thereafter routed along the same path. Since OP_i and OP_j are received at the destination switch in the same pulse, S_{ij} exists. Operation OP_i is emitted by S_{ij} before OP_j by **SW–2(c)** and at subsequent switches, if any, by **SW–2(b)**.

Case 3. Otherwise, OP_i is issued before OP_j by the same PE and the operations are emitted in the same pulse. By **SW–2(a)** and **SW–2(b)**, the operations take the same path to the destination switch and arrive and are executed in the same order in which they are emitted by the SIU. The SIU emits OP_i before OP_j by **SIU–3**.

4.2. Implementing Parops on an Equidistant Network

We consider a network with three kinds of nodes: 1) n nodes containing a PE, an SIU, and a switch (the processor network interface unit); 2) m nodes containing a switch (the memory module interface unit) and a MM; and 3) zero or more nodes containing only switches. The nodes are connected in a ''dance hall'' topology in which every routable path from a PE to an MM is the same length.

These assumptions about the network topology have two consequences that simplify the algorithm for implementing parops:

(1) Since each operation travels the same distance, all operations in any given parop will arrive at their destinations in the same pulse if they are emitted in the same pulse.

(2) Each operation received at a switch in pulse i is emitted in pulse i, not pulse $i+1$. In a general network, each node must emit operations received in pulse i in pulse $i+1$ to avoid deadlock.

The constraints on the SIU and switch algorithms are the same for an equidistant network as for a general network with the following exceptions:

SIU-1′. For any pair of parops, PO_i and PO_j, if PO_i is received before PO_j, then $T(PO_j) \leq T(PO_i)$.

SIU-2′. For any operation, OP_i, OP_i is emitted in pulse $T(parop(OP_i))$.

SW-1′. All operations received in pulse i are emitted in pulse i.

Any algorithm for an equidistant network that conforms with these constraints is consistent with the labeling scheme. Note that **SIU–1′** and **SIU–2′** are equivalent to **SIU–1** and **SIU–2**, respectively, assuming that for any pair of operations, OP_i and OP_j, $d_i = d_j$. The change in **SW–1** affects the consistency proof of **L1** only. For an equidistant network the consistency proof for **L1** is as follows:

L1. By **SW–1′** an operation, OP_i, emitted by a SIU in pulse t is received at the switch in the destination node in pulse t. Since by **SIU–2′**, each operation in $parop(OP_i)$ is emitted by the SIU in pulse $T(parop(OP_i))$ each operation in any given parop is received at its destination in the same pulse, pulse $T(parop(OP_i))$.

4.3. Combining

Parops are consistent with combining. Combining is a technique for maintaining good performance in the presence of multiple operations concurrently accessing the same shared variable. The switches in a combining network fan-in operations that access the same variable and fan-out results to the PE's that issued the combined operations.

The following description of how switches combine operations and decombine responses is adapted from Kruskal, Rudolph, and Snir [KRS88]. Every operation is of the form $(id(OP_i), var(OP_i), op(OP_i))$, where $id(OP_i)$ is a unique identifier for OP_i, $var(OP_i)$ is the shared variable accessed by OP_i, and $op(OP_i)$ encodes an operation and its operands, if any. A switch combines two operations, OP_i and OP_j, that access the same variable by forwarding an operation of the form $(id(OP_i), var(OP_i), op(OP_i) \circ op(OP_j))$, where $op(OP_i) \circ op(OP_j)$ is the composition of $op(OP_i)$ and $op(OP_j)$, and storing $id(OP_i)$, $id(OP_j)$, and $op(OP_i)$. A response corresponding to the combined operation will be of the form $(id(OP_i), val)$. When the response returns, the switch uses the identifier in the response to retrieve the stored information, forwards response $(id(OP_i), val)$ as a reply to OP_i and response $(id(OP_j), op(OP_i)(val))$, where $op(OP_i)(val)$ is the value produced by applying $op(OP_i)$ to val, as a reply to OP_j. Note that each response must return backwards along the path traversed by the corresponding operation.

Each execution of a set of operations using combining is equivalent to some serial execution of the operations. The result of combining OP_i and OP_j as described above is equivalent to a serial execution of the same operations in the order OP_i followed by OP_j. Which of the serial executions is equivalent depends on arbitrary choices made by the switches. If the the switch had reversed the role of OP_i and OP_j, e.g., storing OP_j instead of OP_i, the result would have been equivalent to a serial execution in the reverse order, OP_j followed by OP_i. All associative operations are combinable and unlike as well as like operations can be combined [KRS88].

Operations from parops can be combined if a few rules are observed in addition to the rules that normally govern combining:

C-1. A switch can combine a given pair of operations only if, in the absence of combining, it could emit the operations on the same output line with no token or other operation intervening.

C-2. If a switch combines OP_i and OP_j and, in the absence of combining, the switch could emit the operations on the same output line only if it emitted OP_i before OP_j, then the switch must combine the operations so that the result is equivalent to a serial execution in the order OP_i followed by OP_j.

C-3. Parops can be combined only if the network is ''convex'' and the processor priority function f is a ''convex labeling''. We say that a network is convex if there exists a convex labeling, i.e., an assignment of ID's to the PE's such that the source of all the routable paths through any given switch is a set of PE's with ID's from a contiguous range. A wide class of networks that includes the butterfly and the hypercube is convex. If the processor priority function f is a convex labeling then local combining decisions made by the switches in conformity with **C–1** and **C–2** are guaranteed to be globally consistent with f. If the f function is not a convex labeling, a switch may receive a pair of operations, OP_i and OP_j, where OP_j is a combined operation and OP_i must be executed after the first operation but before the last operation combined to produce OP_j.

Combining parops works, in brief, because **C–1** through **C–3** ensure that for any variable, V, the results of combining the set of operations that access V during any given pulse is equivalent to the result of a serial execution of the same operations in the order induced by the $<_o$ relation. Let $OP_i, OP_j, \ldots, OP_k$, such that $OP_i <_o OP_j <_o \cdots <_o OP_k$, be the sequence of operations that access v during a given pulse. Note that in the absence of combining, any execution consistent with the labeling scheme executes the operations in the correct sequence, the order induced by the $<_o$ relation. Constraints **C–1** and **C–3** ensure that an operation can be combined only with an operation that is adjacent in this sequence or that represents an adjacent operation. Since **C–2** ensures that each pairwise combination is made in the order in which the operations appear within the sequence, the result of combining the operations is equivalent to the result of a serial execution of the operations in the correct sequence.

Ranade proposes an algorithm for combining that is more efficient than previous algorithms in that switches can store the decombining information in FIFO queues instead of in queues requiring associative lookups and less information need be stored [Ran87]. Since the mechanism we use to implement parops is similar to that used by Ranade to allow combining using FIFO queues, we propose using Ranade's algorithm, modified to conform with **C–1** and **C–2**, to implement combining in parops programs. Since Ranade's algorithm requires that the responses be kept sorted in the same order as the corresponding operations, use of the algorithm requires that responses be treated in the same way as operations, except that responses are decombined instead of combined. We are currently exploring generalizations of Ranade's algorithm to networks other than the butterfly.

The implementation of parops does not require a combining network, but parops and combining have a natural affinity in that they are both methods for increasing the concurrency of access to shared memory by adding intelligence to the interconnection network and efficient implementations of both are based on a similar scheme for coordinating the activity of the switches in the network.

4.4. Deadlock

Assuming that the communication network is deadlock-free as well as reliable, the proposed implementation of parops is deadlock-free. Since each MM executes operations in the order in which they arrive, there can be no circular waiting, and therefore no deadlock.

If, however, the communication network is susceptible to deadlock, then so is parops. Unfortunately, the implementation of parops may introduce the potential for deadlock in a communication network that is otherwise free of deadlock. Although Dally and Seitz have proposed a scheme for deadlock-free wormhole routing in arbitrary networks [DaS87], this result is not directly useful here because operations can block not only because a channel required for routing the operation is occupied, but also because of the constraint on the order in which operations arriving on different lines can be emitted on the same line. This constraint, SW–2(c), prevents a switch from emitting an operation, OP_i, waiting at the switch, until it knows that it will not receive an operation on another line that must be emitted on the same line as OP_i and before OP_i. Deadlock is possible because OP_i may directly or indirectly block an operation, OP_j, that must be received at the switch before OP_i can be emitted.

For some networks, deadlock is not a problem. In the reverse baseline and topologically equivalent networks, deadlock in the proposed implementation cannot occur. For each switch at each stage, the inputs to the switch come from disjoint subnetworks. No operation arriving on one input can block another operation from arriving on another input since the operations are in different networks. Since the networks that are topologically equivalent to the reverse baseline include the omega, indirect binary cube, and rectangular SW-banyan [Wu80], deadlock-free implementations of parops using wormhole routing exist for many popular networks.

A general method for eliminating the risk of deadlock created by the ordering constraints is a topic for further research. We are currently working on a method for binary n-cubes. For now, we assume that each switch has sufficient buffer space to store the maximum number of operations it can receive in any given pulse. If c is the maximum number of operations that can be emitted by a PE during a given pulse, $c*n*b$ bytes, where b is the maximum size of any operation, is an upper bound on the size of the buffer space. The buffer space allows switches to use virtual cut-through, removing waiting operations from the network, so that they do not block other operations.

Assuming deadlock-freedom, any implementation that is consistent with the labeling scheme is fair. If every PE issues a finite number of operations in any given pulse and there are a finite number of PE's, then for any given operation, OP_j, there are a finite number of operations, OP_i, such that $OP_i <_o OP_j$, Thus, by **L3**, a finite number of operations that can be executed before OP_j.

5. Applications

Parops are intended to be used as a primitive from which to construct more powerful mechanisms, but they have some direct applications as well. In this section we will show examples of the use of parops, give a brief and informal description of how parops can be used to support a method for coordinating access to shared memory, and describe some applications for local synchrony other than in the implementation of parops.

5.1. Applications for Parops

The problem that originally suggested parops is that of maintaining reference counts in a massively parallel linked data structure where garbage collection of deleted nodes is by reference counts. A process traversing a link in the structure must read a pointer and increment the reference count associated with the pointer as an atomic action:

```
lock(POINTER);       lock(REF_COUNT);
read(POINTER,v);     inc(REF_COUNT);
unlock(POINTER);     unlock(REF_COUNT);
```

Assuming the memory modules support atomic execution of the RMW instruction `inc`, a parop can read the pointer and increment the associated reference count as an indivisible step [Wag87]:

```
POINTER:read(v) || REF_COUNT:inc;
```

Besides reducing in the number of instructions executed by each process, the use of parops increases the concurrency of the solution.

A way to use parops in chasing pointers (or using a shared index into a shared array) is the ''repeated read'' technique. A process using this technique first reads the pointer and then issues a parop that both dereferences and rereads the pointer. The second read confirms that the value of the pointer read initially is still valid when the value is used. If the values returned by the two reads differ, the process tries again until it executes two consecutive reads of the pointer that return the same value. Note that the dereferencing operation must, in general, be a read. If the repeated read fails, any write based on the invalid value for the pointer updates the wrong node. Another process may see the result of the misdirected write before it can be cancelled.

5.2. Parops as a Basis for Implementing Atomic Actions

The principal application for parops is as a synchronization primitive on which to build other mechanisms. Parops are not very useful by themselves because of two limitations. First, every operation in a parop must be independently issuable, but a process does not, in general, have enough information to issue all of the operations required for a given atomic action before any operation completes. Second, any parop that contains an operation on a variable that may be locked must contain only that single operation. If a parop contains an operation that accesses a locked variable, either the access is allowed, in violation of the lock, or it is blocked. If the access is blocked, the semantics of parops is violated since other operations in the same parop that access unlocked variables will be executed. Because of these limitations, we do not propose a new parallel language based on parops.

In a companion paper we propose using parops as one of the bases for a general method for implementing atomic actions and process synchronization [WiR89]. This proposal is based on two ideas in addition to parops:

(1) Variables can be represented as a sequence 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 the context for an access by reserving a position in the variable's access sequence. Note that the scheduling step for a write can be executed before the value to be

written has been determined. The assignment step 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 process when it determines the value to be assigned. The assignment step for reads 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.

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. The use of the parop in the scheduling step reserves a consistent ''slice'' across the sequences of the shared variables accessed in the atomic action.

5.3. Applications for Local Synchrony

Local synchrony has applications other than in the implementation of parops. It can support an efficient implementation of combining [Ran87], can simulate a synchronous network on an asynchronous network so that SIMD graph algorithms can be executed [Awe85], and can be used to emulate a PRAM on an asynchronous multiprocessor [RBJ88].

Another application of local synchrony related to its ability to emulate a synchronous machine is in debugging parallel programs. If each SIU emits exactly one operation per process per pulse, the machine operates in ''synchronous mode''. Note that in synchronous mode a program will execute deterministically, always producing the same result given the same input and initial state. An error that occurs in the normal asynchronous mode but not in synchronous mode is a time dependent error resulting from an incorrect interleaving of accesses by processes executing at different rates.

Finally, local synchrony is a mechanism that supports correct, i.e., sequentially consistent, pipelining of memory accesses. We have shown that parops can be pipelined without sacrificing sequential consistency. Since a parop, in the trivial case, is a single operation, this benefit extends to ordinary programs that are executed on a machine that supports parops.

Maintaining sequential consistency is a hard problem in an asynchronous computation because stochastic delays in the interconnection network allow different operations issued by the same process to arrive at the memory modules in an order that is inconsistent with the order in which the operations were issued. The most straightforward solution is for each process to delay issuing an operation until it receives notice that execution of its preceding operation is complete. This solution has two drawbacks: (1) it prohibits processes from decreasing the effective latency of the network by pipelining operations; and (2) it is overly inclusive — some operations may be pipelined without introducing any risk of violating sequential consistency. Lamport proposes a technique that eases the restriction on pipelining by permitting a process to issue an operation as soon as it is notified that its last operation is queued awaiting execution at the memory module [Lam79]. On some machines a special ''fence'' instruction is available which the programmer can use to indicate that the next memory access cannot be issued until all memory accesses previously issued by the process have been executed [KRS88]. Shasha and Snir describe an algorithm for statically analyzing programs to determine which operations can be safely pipelined [ShS88]. On a machine that supports parops, all operations can be pipelined. Assuming that operations are issued in the order specified by the program, the execution will be sequentially consistent.

These applications demonstrate that local synchrony has generality. For the price of parops, the machine designer gets a machine that can pipeline memory accesses and simulate SIMD algorithms (without requiring the insertion of barriers after each global reference), a debugging tool, and the basis for an efficient combining network.

6. Concluding Remarks

There are several directions in which the research described in this paper can be extended. We describe a few of these below:

Detailed design and performance studies. Our focus in describing algorithms for the SIU's and switches has been on correctness. We intend to identify design alternatives for the switches and SIU's and to evaluate their performance. Some design decisions that may have a significant affect on performance include the following:

1) The flow control protocol, especially between each SIU and its associated switch, and the related question of the ''granularity'' of a pulse, the minimum time an SIU waits after emitting a token before emitting the next token.

2) The information carried by a token. If a token carries information about the operations that follow, the information may enable the switch to route operations on all outputs concurrently.

3) Additional control messages. Ranade proposes the use of ''ghost messages'' to improve the performance of the combining network [Ran87]. When a switch emits a message on one output line it sends a copy, a ghost message, on every other output line. The receipt of a ghost message tells a switch that no message that precedes the ghost message in sorted order will be received on that line. This information may allow the switch to emit operations received on other lines earlier than it could in the absence of the ghost message. Ghost messages may also be useful in a network that supports parops.

Deadlock-free routing and combining of operations. We are currently studying extensions of Ranade's algorithm to networks other that the butterfly, looking for algorithms that are deadlock-free and consistent with both wormhole routing and parops.

Fault-tolerance. The parop implementation we have described has two characteristics that may provide a basis for a fault-tolerant implementation. First, the implementation imposes a regular pattern of communication, the token pulses, on the network. The switches may be able to use these pulses as a heartbeat to diagnose failure points. Second, the implementation allows multiple paths on which to route any given operation. If one path has a malfunctioning link or switch, there may be an alternative path. An efficient fault-tolerant implementation that also supports combining may be more difficult to find since the rule in combining networks is that the response to an operation must take the same path as the operation since decombining information is stored at switches along that path. A solution we intend to investigate is to double the network, pairing each node with its copy. If each node pair shares combining information, a response can return through either node in a given node pair.

Extension to the Message Based Model. We envision using parops in the MBM to implement lock-free access to distributed objects, i.e., objects that are distributed over more than one multicomputer node. An important question to explore is whether parops can be nested to facilitate information hiding and object-oriented programming.

Hybrid synchronous-asynchronous operation. In the section on applications for local synchrony we proposed that a machine that supports parops could operate in synchronous mode. An ability to specify a subset of the SIU's to operate in synchronous mode suggests the possibility of mixing synchronous and asynchronous computation within the same program. Subsets of processes could execute as teams, each executing a program from a library of synchronous programs called as subprocedures from an asynchronous program. One question to explore is the performance of such hybrid computations.

Application to the cache coherency problem. We are studying the use of parops in solving the cache coherency problem and its dual in the MBM, the virtual memory problem. We believe that parops may give an efficient solution to these problems because parops provide a way to update multiple copies of a variable atomically without locking.

References

[Awe85] B. Awerbuch, Complexity of Network Synchronization, *J. ACM 32*, 4 (October 1985), 804-423.

[DaS87] W. J. Dally and C. L. Seitz, Deadlock-Free Message Routing, *IEEE Trans. on Computers 36*, 5 (May 1987), 547-553.

[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.

[Lam79] L. Lamport, How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, *IEEE Trans. on Computers 28*, (1979), 690-691.

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

[Pap86] C. Papadimitriou, *Database Concurrency Control*, Computer Science Press, 1986.

[Ran87] A. G. Ranade, How to Emulate Shared Memory, *IEEE Annual Symp. on Foundations of Computer Science* , Los Angeles, 1987, 185-194.

[RBJ88] A. G. Ranade, S. N. Bhatt and S. L. Johnson, The Fluent Abstract Machine, Tech. Rep. 573, Yale University, Dept. of Computer Science, January, 1988.

[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.

[SiK80] A. Silberschatz and Z. Kedem, Consistency in Hierarchical Database Systems, *J. ACM 27*, 1 (January, 1980), 72-80.

[Wag87] R. R. Wagner, Jr., *Parallel Operations in Shared Memory*, Master's Thesis, University of Virginia, Charlottesville, Virginia, August, 1987.

[WiR89] C. Williams and P. F. Reynolds, Jr., On Variables as Access Sequences in Parallel Asynchronous Computations, Tech. Rep. 89-17, University of Virginia, Department of Computer Science, December, 1989.

[Wu80] C. Wu and T. Feng, On a Class of Multistage Interconnection Networks, *IEEE Trans. on Computers 29*, 8 (August 1980), 694-702.