COMBINING ATOMIC ACTIONS IN A RECOMBINING NETWORK

Craig Williams
Paul F. Reynolds, Jr.

Computer Science Report No. TR-91-33
November, 1991

Combining Atomic Actions in a Recombining Network

Craig Williams
Paul F. Reynolds, Jr.

Abstract: Recombining networks have been proposed for the purpose of increasing the concurrency of accesses to shared variables and reducing the incidence of hotspots within the network. A recombining network recursively combines concurrently issued operations on the same variable and then fans out responses to the processes that issued the operations. We describe the *isotach* network, a recombining network similar to a network proposed by Ranade, and the *isochron*, a logically synchronous multicast based on the isotach network. The isochron directly supports a limited class of atomic actions, called *flat* atomic actions, in which all the operations can be issued as a batch. We show operations from different isochrons that access the same variable can be combined within the interconnection network consistently with the semantics of isochrons. The ability to combine isochrons means operations on the same variable can be combined even though they come from different atomic actions. Previous work on recombining networks concerns only individual operations on shared variables.

1. INTRODUCTION

This paper describes how simple atomic actions can be combined within the interconnection network (ICN) of a multiprocessor. ICN's that can combine operations, called combining or recombining networks, have been built or proposed for several machines, including the NYU Ultracomputer [GGK83], IBM's RP3 [Pfi85], and the Yale Fluent [RBJ88]. The second generation Connection Machine [TuR88] has a limited recombining network, capable of combining oprations but not of returning intermediate results. The switches in a recombining network fan-in concurrently issued operations on the same variable and fan-out responses to the processes that issued the operations.

Combining is a technique for maintaining good performance in the presence of multiple operations concurrently accessing the same shared variable. On a multiprocessor with a conventional network, multiple concurrent operations on the same variable are executed serially at the memory module (MM) containing the variable. If each operation is a read or write (or other associative operation, as described in section 4), a recombining network can combine these operations so that the MM need execute only the resultant composite operation. Assuming a network traversal time of $O(\log n)$, where n is the number of processing elements (PE's), combining reduces the time for executing r combinable operations from $O(\log n)+O(r)$ to $O(\log n)+O(1)=O(\log n)$. Recombining networks also reduce network load and may reduce the incidence of hot-spots [PfN85].

We explore combining operations that are part of different non-trivial atomic actions, i.e., atomic actions that access more than one shared variable. This paper shows operations can be combined even though they are from different, non-trivial atomic actions. This result is limited to a restricted type of atomic action we call a *flat* atomic action. In a *flat* atomic action no operation depends on another operation in the same atomic action, i.e., all the operations can be issued as a batch.

Our proposal for combining operations from different atomic actions requires a recombining network that is also an *isotach* network [RWW89]. An isotach network implements a logical time system that relates communication time to communication distance. Each message in an isotach network progresses towards its destination at the same rate: one switch per logical time unit. This characteristic property of the isotach network, called the *velocity* invariant, makes the network a powerful coordinating mechanism. We have used the isotach network as the basis for a logically synchronous multicast [RWW89], for concurrency control techniques that work without operations on locks [WiR89], and for a new family of highly concurrent cache coherence protocols [WiR90].

The isotach networks describe in this paper are similar to a network proposed by Ranade [Ran87, RBJ88]. Ranade uses a network conforming to our definition of an isotach network to support efficient emulation of a concurrent-read concurrent-write parallel random access machine (CRCW-PRAM). We defined the isotach network independently in a different context, concurrency control in asynchronous computations, and we have abstracted the useful properties of the network from the particular application. Ranade's work is important to this research both as an early use of the isotach network for combining and as the basis for efficient combining switches. We return to Ranade's algorithm below.

We show the isotach network supports a logically synchronous, sequentially consistent multicast called the *isochron*. Operations in an isochron are executed atomically, i.e., they appear to be executed as an indivisible step. Thus the isochron is a limited form of atomic action, limited because the process must issue all the operations in an isochron as a batch. Execution of isochrons is also sequentially consistent, i.e., isochrons issued by the same process appear to be executed in the order in which they were issued. We show combining preserves the atomicity and sequential consistency of isochrons.

An isotach network supports isochrons by ensuring operations from different isochrons are executed on all shared variables in a consistent order. Given this approach to ensuring atomicity, it is easy to see how atomicity can be preserved by a recombining network that is also an isotach network. The switches combine operations so that execution of the composite operation is equivalent to execution of the constituent operations in this original, consistent order.

We described isochrons and the isotach network in previous reports [RWW89, WiR89, WiR90]. For completeness, we summarize these descriptions below, in sections 2 and 3. Section 4 describes previous work on recombining networks. Section 5 shows how operations can be combined in an isotach network consistently with the semantics of isochrons.

2. ISOTACH NETWORKS

Isotach networks are based on a form of synchronization we call *local synchrony*. In a locally synchronous system, each network node keeps its logical clock loosely synchronized with the logical clocks of its neighbors. In more vivid terms, the network pulses like a heart. These pulses supply the timing mechanism for a distributed logical clock. Local synchrony has been used by Ranade in a CRCW-PRAM emulation, by Awerbuch to support execution of SIMD graph algorithms on asynchronous networks [Awe85], and by Birk, et al., to support barrier synchronization [BGS89].

Isotach networks can be implemented on a variety of topologies and for many different purposes. For simplicity, we restrict this discussion to the traditional context for discussing recombining networks: shared memory model (SMM) computations on equidistant networks such as the network shown in Figure 1. In an equidistant network, sometimes called a *dance hall* network, the length of every path from a PE to an MM is the same. Figure 1 shows an equidistant network with reverse baseline topology. This network, under renumbering of inputs and outputs, is equivalent to many other common networks including the omega and indirect binary cube [Wu80]. Each link illustrated in Fig. 1 is bidirectional, i.e., it represents two channels, one in each direction. We require that each channel be FIFO. For simplicity, we assume each element, i.e., each PE or MM, is connected to the network via a switch interface unit (SIU) and each process executes on its own PE.

We assume processes communicate only by accessing shared memory (we ignore interrupts). An *operation* is an instruction accessing a shared variable and a *response* is the reply to an operation. A *message*, i.e., an operation or a response, is *emitted* when the SIU for the source element submits it to the ICN and is *received* when the SIU for the destination element accepts it from the ICN.

With each SIU we associate a local logical clock, a function that assigns monotonically increasing times to local emit and receive events. In general, each logical time is an ordered pair

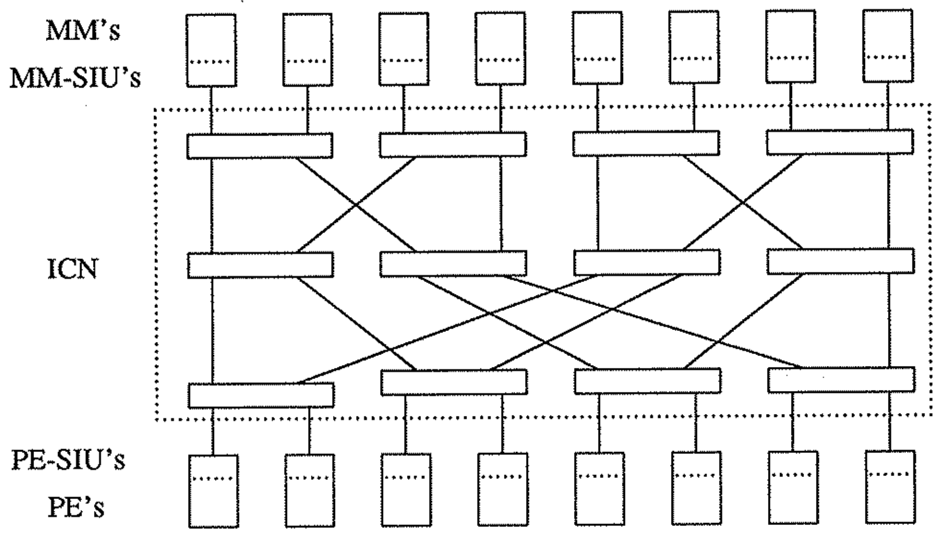

Figure 1. An Equidistant Network

(*pulse*, *time_within_pulse*), where the first component is an integer and the second an n-tuple of integers. In the logical time system used in this paper, each logical time is a 4-tuple of integers, where the first is the pulse component. Logical times can be compared component-wise and are lexicographically ordered. Logical time can be elastic in relation to physical time, as it is in the implementation described below. A unit of logical time is not necessarily of any fixed duration according to physical time.

In an isotach network each message is received exactly *DIST* pulses after it is emitted, where *DIST* is the number of switches through which the message is routed, i.e., a message emitted at time t_{emit} = (i,v,j,k) is received at time $t_{receive}$ = $(i+DIST,v,j,k)$. An isotach network maintains the following *velocity invariant*:

$$DIST \,/\, (t_{receive} - t_{emit}) = 1 \; switch/pulse$$

All messages in an isotach network travel at the same velocity in logical time — one switch per pulse. The time at which a message is received in an isotach network is thus completely determined by the time at which it is emitted. Note, however, that though *logical* message delivery times are deterministic, *physical* delivery times are not.

This logical time system has precursors in the system of logical clocks proposed by Lamport [Lam78] and extended by Mattern [Mat88] and Fidge [Fid91]. These other logical time systems are designed to assign times that are consistent with potential causality, i.e., if event *a* can affect event *b* then the time assigned to event *a* is less than the time assigned to event *b*. The system of logical time realized by the isotach network differs from these other logical time systems in that events are assigned times consistent with both causality and the velocity invariant.

An isotach network can be implemented on a network with FIFO communication links between neighboring switches and elements. Switches exchange control signals called *tokens* with neighboring switches. Initially each switch emits a token pulse, i.e., it emits a token on each output. After the initial pulse, each switch emits token pulse *i* after receiving token *i*−1 on all inputs. Thus each switch is loosely synchronized with its neighbors. The token pulses also drive the clocks at the SIU's. In each token pulse, a switch emits a token on each output, including the output to each adjacent SIU, if any, and it receives a token on each input, including the input from each adjacent SIU, if any, before emitting the next token pulse. When it receives token i from the ICN, the SIU sets its local clock to $(i,0,0,0)$ and sends the token back to the ICN. The pulse component of the time at each SIU is the number of tokens that have passed through the SIU.

Between tokens, the SIU for a PE (PE-SIU) may emit zero or more operations. Before emitting each pulse of operations, a PE-SIU first sorts the operations in increasing order by address of the variable accessed, using a stable sort to preserve the issue order among operations on the same variable. Each operation receives a timestamp. The timestamp for operation OP_i, denoted $ts(OP_i)$, is the 4-tuple $(pulse, var, pId, rank)$, where $pulse$ is the pulse in which OP_i is emitted by the PE-SIU, var is the variable accessed, pId is the identifier of the issuing PE, and $rank$ is OP_i's issue rank within the pulse, i.e., the rank component of the timestamp for OP_i is j if OP_i is the jth operation emitted by that PE-SIU in the current pulse. Note that a PE-SIU emits operations in timestamp order and that timestamps are unique. This technique for assigning unique timestamps without centralized control is widely used in database concurrency control [Ree83,RSL78]. When a PE-SIU emits an operation it updates the local clock, setting the clock equal to the timestamp of the operation. Each PE-SIU emits operations in timestamp order so each PE-SIU's clock moves forward monotonically.

Between token pulses, each switch routes operations as usual except it chooses messages to route in timestamp order. A switch with j inputs and k outputs is continuously merging the j sorted lists arriving on its inputs to produce k sorted output lists. When it routes a message, the switch increments the pulse component of the message's timestamp. Since each SIU emits operations in timestamp order and timestamp order is maintained at each switch and across each link, operations are received at each MM in timestamp order. Consider the tree of switches rooted at a given MM with leaves at each PE-SIU. A simple induction on the depth of the tree shows that operations arrive at the root MM in strictly increasing order by timestamp. (See lemma 2 in section 5.4.)

Each SIU for an MM (MM-SIU) maintains a local clock in the same way as a PE-SIU, except an MM-SIU updates its clock for receive events. When it receives an operation, an MM-SIU sends the operation to the associated MM and emits the response before receiving the next operation. Thus the execution time of operation OP_i and the emit time t_{emit} of the response to OP_i both equal $t_{receive}$ for OP_i. Each response carries a timestamp and is returned to the PE in order by timestamp in the same way that operations are delivered to the MM's in order by timestamp. Since the response to operation OP_i inherits its timestamp from OP_i, responses are received at each PE in the order in which the corresponding operations were emitted.

The velocity invariant holds because a message with timestamp (i,v,j,k) arriving at a switch in pulse i (after the ith token received on the input on which the message arrives) leaves with timestamp $(i+1,v,j,k)$ in pulse $i+1$ (after the $i+1$st token pulse). Since traveling through a switch adds 1 to the pulse component of a message's timestamp and does not otherwise change the timestamp, a message emitted at time (i,v,j,k) is received at time $(i+DIST,v,j,k)$.

The implementation described here is an abstract implementation intended to be useful in reasoning about the isotach network, but is not an implementation we recommend for an actual system. Though the tokens are necessary, the timestamps and logical clocks are not. Operations need carry only the information they carry in conventional networks. To decouple the routing of operations from the routing of responses and to make wormhole routing possible, an actual implementation would also divide this network of bidirectional channels into two virtual networks of unidirectional channels, one for operations and the other for responses.

3. ISOCHRONS

The isotach network supports a logically synchronous multicast called the *isochron*. An isochron is a group of operations issued by the same process where execution is atomic and sequentially consistent. The operations in an isochron must be issued as a batch. The isochron can thus also be described as a sequentially consistent, *flat* atomic action.

The atomicity of isochron execution means operations in an isochron appear to be executed simultaneously, i.e., as an indivisible step. Typically, atomicity is ensured by obtaining locks on the accessed variables, although locking heavily accessed variables lessens concurrency and may create bottlenecks. The isotach network ensures atomicity on multiple shared variables *without* requiring locks. Operations in an isochron are executed atomically without obtaining exclusive access rights to the accessed variables beyond that implied by the MM hardware. Assuming an MM can execute only one operation at a time, each operation, whether it is part of an atomic action or not, effectively locks the MM. In a combining network, even this MM-level source of serialization can be eliminated to the extent it is created by operations on the same variable.

Sequential consistency means the order in which accesses are executed is consistent with the order specified by each individual process's sequential program [Lam79]. Maintaining sequential consistency is a problem in multiprocessors because stochastic delays in the ICN allow operations issued by the same process to arrive at the MM's in an order inconsistent with the order in which the operations were issued. The simplest solution, disallowing pipelining of memory accesses, is undesirable since pipelining is an important way to lessen effective memory latency. An isotach network allows operations to be pipelined while ensuring operations within each isochron appear to be executed in the order specified and isochrons issued by the same process appear to be executed in the order in which they are issued.

Syntactically, an isochron is a list of one or more operations terminated by a semicolon in which adjacent operations are separated by double bars "||." For example, in the following code segment

```
read(A,a) || read(B,b);
write(A,b);
```

the first isochron assigns the value of shared variables A and B to local variables a and b, respectively. The second isochron, a singleton isochron containing only one operation, assigns A the value returned by the read on B in the first isochron. Execution is sequentially consistent in that the read operation on A is executed before the write on A and is atomic in that the first isochron returns a consistent view of A and B. If a second process concurrently executes the isochron `write(A,5) || write(B,5);` the first process will read either the old value, before execution of the second process's writes, for both A and B or the new value for both A and B.

Note that atomic execution of the assignment `A := B`, where A and B are shared variables, cannot be expressed as an isochron, i.e., the assignment is not a flat atomic action. Execution of this assignment requires two operations, a read to B and a write to A, but the operations cannot be issued as a batch because the write cannot be issued until after the read is executed. Despite this limitation, isochrons are powerful enough to allow processes to obtain a consistent view of multiple shared variables and to make consistent updates.

Implementation of isochrons is based on the velocity invariant. If a PE-SIU knows *DIST*, the distance an operation travels to memory, the PE-SIU can control $t_{receive}$ by its choice of t_{emit}. To ensure atomic, sequentially consistent execution of isochrons, each PE-SIU emits operations issued by the associated process in accordance with the following emission rules:

ATOMICITY. Emit operations from the same isochron so they are received (and thus executed) in the same pulse.

SEQUENTIAL CONSISTENCY. Emit each operation so it is received in a pulse equal to or greater than that of the operation issued before it.

For equidistant networks, application of the emission rules is especially simple. A PE-SIU emits all operations from the same isochron in the same logical pulse and emits operations in the order in which they are issued by the associated process.

Execution of isochrons is atomic and sequentially consistent because parallel execution of a program on an isotach network is equivalent to a serial execution in which 1) operations in each isochron are executed without interleaving with operations from other isochrons, and 2) operations issued by the same process are executed in the order in which they were issued. Two executions of the same program are *equivalent* if every variable is accessed by the same operations and conflicting operations are executed in the same order in both executions, where operations conflict if they access the same variable and at least one is a write [Pap86]. Since in a parallel execution on an isotach network operations on each variable are executed in order by the first, third, and last components of their timestamp, execution is equivalent to a serial execution in which the same operations are executed in order by the first, third, and last components of their timestamp. Thus isochrons emitted in different pulses appear to be executed in order by the pulse component, isochrons emitted in the same pulse by different PE's appear to be executed in order by the *pId* of the issuing PE, and operations issued by the same PE in the same pulse appear to be executed in the order in which they are issued.

4. RECOMBINING NETWORKS

Assume several PE's concurrently issue operations accessing a given shared variable v located in MM_i. A recombining network avoids serial execution of these operations at MM_i by using the tree of switches rooted at MM_i to fan-in operations on v and fan-out results to the PE's that issued the operations. When two operations on the same variable collide at a switch, the switch combines the operations and forwards a single composite operation that has the same effect as a serial execution of the operations from which it was created. The switch also stores information for use when the result of the operation returns to the switch from memory. The need to fan-out results imposes the requirement that each response must return backwards along the path traversed by the corresponding operation. The stored combining information, together with the value returned from memory, allows the switch to compute responses for each of the combined operations. Combining is recursive, i.e., an operation combined at one stage may itself be the product of combining at a previous stage.

We consider associative Read-Modify-Write (RMW) operations [KRS88], also known as a $fetch\text{-}and\text{-}\phi$ operations [GGK83]. An RMW operation indivisibly reads a shared variable, assigns it a new value that is a specified function of the old, and returns the old value. Reads and writes, as well as swaps, test-and-sets, and fetch-and-adds, are all special cases of the RMW and are all combinable. For a read, the specified function is the identity function, $f(x)=x$, so that the read assigns the old value of v as the new value of v. For a write, the function is the constant function, $f(x)=c$, where c is the value supplied by the write as an operand, and the returned value is ignored. A swap is a write in which the returned value is used. In many cases *unlike* as well as *like* RMW operations can be combined. In particular, read, write, and swap operations on the same variable are combinable.

Let OP_i be an associative RMW, f_i be the function specified by OP_i, including any operands supplied by OP_i, and id_i be a tag that uniquely identifies OP_i. Assuming the operations are combinable, a switch can combine OP_i and OP_j by forwarding a composite operation OP_{ij}, where $f_{ij} = f_j \circ f_i$, the composition of f_j and f_i, Note that $f_j \circ f_i$ designates the execution of f_i before f_j, i.e., $f_j(f_i(\text{v}))$, where v is the variable accessed by OP_i and OP_j. When the response to OP_{ij} returns to the switch from memory, the switch must send responses to both OP_i and OP_j. To enable it to do so, the switch must remember id_j and f_i. In a standard recombining network, the switches use associative lookup queues to store this combining information. The switch stores id_j and f_i, with id_i as the associated key. Assuming OP_{ij} reaches memory without being further combined and that $\text{v}=val$ when OP_{ij} is received, $\text{v} = f_j(f_i(val))$ after execution of OP_{ij} and val is returned to the switch in response to OP_{ij}. When the response reaches the switch, it retrieves the combining information, id_j and f_i, from its associative lookup queue using the tag of the response it receives from memory, id_i, as the search key. The switch can then send $f_i(val)$, tagged with id_j, as the response to OP_j and val, the old value of v, tagged with id_i, as the response to OP_i. Thus the results of executing OP_{ij} are the same as the results of executing OP_j after OP_i.

When a switch combines a pair of operations, OP_i and OP_j, it gives the resultant composite operation, OP_{ij} or OP_{ji}, an *orientation*. Operation OP_{ij} is equivalent to OP_i followed by OP_j. The operation with the reverse orientation, OP_{ji}, is equivalent to execution of OP_j followed by OP_i. The orientation of the operation that results from combining OP_i and OP_j depends on which information the switch forwards and which it stores when it combines OP_i and OP_j. The switch forms OP_{ij} as described above. It forms operation OP_{ji} by storing id_i and f_j with the key id_j and forwarding an operation tagged id_j that specifies the function $(f_i(f_j(\text{v}))$. In general, any pair of operations that can be combined can be combined in either orientation [KRS88]. A read and a write can be combined, in that order, by forwarding a swap that assigns the value supplied by the write and the read is satisfied by the value returned by the swap. The read and write can be combined in the reverse orientation, i.e., as write followed by a read, by forwarding the write. Memory need not return a value because the switch can satisfy the read with the value supplied by the write.

Kruskal, Rudolph, and Snir prove this recombining network algorithm is correct by showing that the result of executing each composite operation OP_c is equivalent to the result of a serial execution of the operations OP_c *represents* [KRS88]. An original operation, i.e., an operation issued by a process, represents itself and a composite operation produced by combining OP_i and OP_j represents the operations OP_i and OP_j represent. They prove furthermore that execution of OP_c is equivalent to not just any serial execution, but to the serial execution that reflects the orientation of the original operations in OP_c. If $f_c = f_n \circ f_{n-1} \circ, \ldots, \circ f_1$, then execution of OP_c is equivalent to the serial execution of $op_1, \ldots, op_n$, in that order. In other words, execution of OP_c is equivalent to serial execution of the operations OP_c represents in the reverse of the order in which the functions they specify appear in the composition of functions represented by f_c[1]. We use this result below in proving the algorithm for combining atomic actions.

Ranade [Ran87] shows that associative lookup queues located at the switches can be replaced by simple FIFO's assuming the recombing network is essentially what we call an isotach network. A FIFO is sufficient on an isotach network because the velocity invariant ensures responses return to a switch in the same order the corresponding operations leave the switch. When a response returns to a switch from memory, the combining information will be at the head of the FIFO. Ranade also shows that an isotach-like recombining network need store less combining information. Instead of the identifying tags, the switch can store two direction bits indicating on which outputs the switch should return responses.

Though not noted by Ranade, isotach recombining networks support a combining short-cut not supported on conventional recombining networks. When a write is combined with a read in an isotach recombining network the value written by the write (assuming its timestamp is less than the read's) can be returned immediately as the response to the read. In a conventional recombining network, the read must be delayed at the switch where it was combined until a value returns from memory in response to the forwarded write. Returning the read immediately can cause a violation of sequential consistency. If the forwarded write is delayed in the ICN, a second read to the same variable that logically succeeds the write may be executed before the write [DuS90, KRS88]. On an isotach network the read can be satisfied immediately because any operation on the same variable that logically succeeds the write will have a greater execution timestamp and so will be executed after the write. The short-cut comes at a cost — it is inconsistent with the simple FIFO storage of combining information.

5. COMBINING ATOMIC ACTIONS

We show that operations can be combined within an isotach network in a manner consistent with the semantics of isochrons. This means operations from different flat atomic actions can be correctly combined in the ICN of a multiprocessor, increasing the concurrency of access to shared variables over that previously possible. This section describes the algorithm for combining operations in an isotach network and proves the correctness of the algorithm by showing it does not change the order in which

[1] This statement of the result differs from that in original paper [KRS88] because we use $f_i \circ f_j$ to denote $f_i(f_j(x))$ whereas in the original paper it denotes $f_j(f_i(x))$.

operations are effectively executed.

5.1. Convexity

We consider convex isotach networks. A network is *convex* if

(1) The routable paths for each shared variable v form a tree, called v's *routing tree*, with the MM containing v at the root, a PE-SIU at each leaf, and an ICN switch at each interior node; and

(2) For each variable v, and for each interior node S in v's routing tree, there is an interval of *pId*'s $[i..j]$, $i \leq j$, such that for all k, $i \leq k \leq j$, all operations (in a combining network, all operations representing operations) on v from PE_k go through S. This interval is S's *pId−interval* for v.

A *routable* path is a path consistent with the network's routing algorithm. Depending on the routing algorithm, some physical paths may not be routable paths. The reverse baseline network (see Fig. 1) is convex as are all networks equivalent to the reverse baseline network under renumbering of inputs and outputs. Some non-convex networks can be made convex by changing the routing algorithm or the *pId* assignment, but some networks, such as the network in Figure 2, are inherently non-convex. The 1st PE numbering shown is satisfactory for V_1, but not for V_2, and the 2nd for V_2, but not for V_1. No numbering satisfies the second convexity condition for both V_1 and V_2.

Convexity is important because it allows switches to make local decisions about whether operations can be combined. In a non-convex isotach network, a switch S cannot decide whether OP_i and OP_j can be combined safely because there may be an intervening operation OP_x, i.e., an operation with a timestamp between that of OP_i and OP_j, that does not go through S. If S combines OP_i and OP_j anyway, a later switch may receive both OP_{ij} and OP_x. Since combining is irreversible after the combined operation leaves the the switch where it was created, a later switch cannot correct the error. In a convex network a switch S can safely decide whether to combine operations from the same pulse on the same variable because all operations with intervening timestamps also go through S in the same pulse.

5.2. The Combining Algorithm

On an isotach combining network, the timestamp of each operation is not a single logical time but an *interval* of logical time. The timestamp for any operation OP_i is an ordered pair (t_{ib}, t_{ie}), $t_{ib} \leq t_{ie}$. If OP_i is an original operation, $t_{ib} = t_{ie}$ and t_{ib} is the timestamp OP_i would receive in a non-combining isotach network, as described in section 3. Operation OP_{ij} formed by combining OP_i, with timestamp (t_{ib}, t_{ie}), and

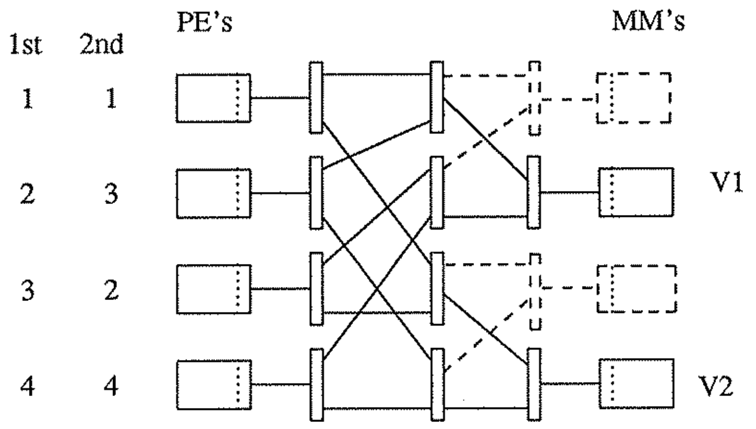

Figure 2. A Non-Convex Network

OP_j, with timestamp (t_{jb}, t_{je}), with the orientation OP_i followed by OP_j, has timestamp (t_{ib}, t_{je}).

Each switch S routes operations using the same rule as in a non-combining isotach network (section 2), adapted to accommodate the different timestamp format: S routes OP_i before OP_j if $t_{ib} < t_{jb}$. (Later we show $t_{ib} < t_{jb} \Rightarrow t_{ie} < t_{jb}$, i.e., that timestamps of different operations that go through the same switch specify disjoint ranges of logical times.) Switch S combines operations under the same conditions and in the same way as in other combining networks (section 4) except

(1) Operations can be combined only if they arrive at S in the same pulse and only if, in the absence of combining, S would emit them one after the other, with no intervening operation. If S combines OP_i and OP_j, there is no operation OP_x, such that OP_x goes through S and $t_{ib} < t_{xb} < t_{jb}$. Since any operation such as OP_x must arrive in the same pulse and access the same variable as OP_i and OP_j, compliance with this rule is easily ensured.

(2) If S combines OP_i and OP_j, it creates OP_{ij} if $t_{ib} < t_{jb}$ and OP_{ji} otherwise. (It will be shown that $t_{ib} = t_{jb}$ cannot occur.) The orientation is important. Changing the orientation changes the effective order in which operations are executed and could cause operations from different isochrons to be executed in an inconsistent order at different variables. Requiring that switches combine operations in order by the pId's of the issuing PE's has been proposed by other researchers as a way to compute parallel prefixes within the ICN [KRS88, RBJ88].

We require operation OP_{ij} formed by combining OP_i and OP_j be routed on the same path OP_i would take if the combining had not occurred. Since the routing paths for v form a routing tree, this path is also the same as the path for OP_j. This requirement means that any operation representing any original operation OP_i takes the same path OP_i would take in the absence of combining. Thus combining preserves convexity.

Figure 3 shows the resultant timestamps and functions produced by combining six concurrently emmited operations on the same variable v. The operation with timestamp ((4,v,0,1),(4,v,6,3)) executed at

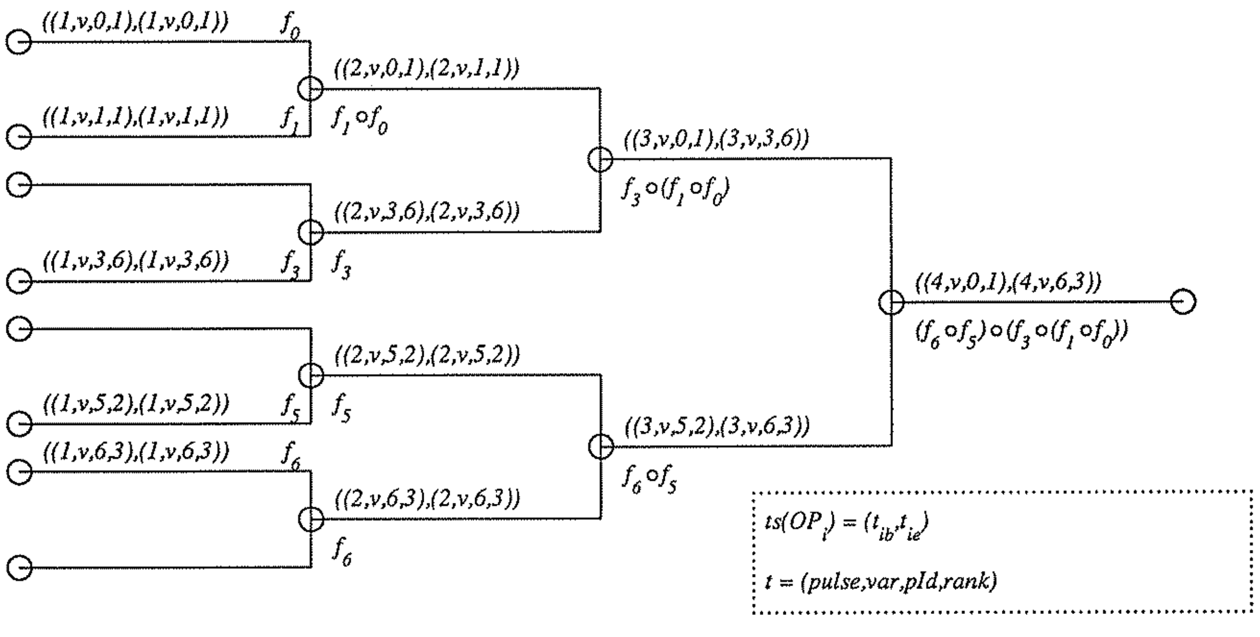

Figure 3. Combining Operations on v

memory represents all six operations. Note that the functions specified by the original operations appear in the composition of functions specified by this composite operation in reverse order by timestamp of the original operation.

5.3. Proof Preliminaries

Intuitively, combining works because it preserves the effective execution order of operations on the same variable. Before proving the isotach combining algorithm, we describe our notation, define a few terms, and list some basic properties of isotach combining networks.

The relations *include*, *disjoint*, *overlap*, *less than* ($<_{[]}$), *greater than* ($>_{[]}$), and *equals* ($=_{[]}$), over intervals where the starting and ending values are integers or lexicographically ordered integer tuples are defined as the reader would expect: for any two such intervals, $INV_i = [v_{ib} \,..\, v_{ie}]$, $v_{ib} < v_{ie}$, and $INV_j = [v_{jb} \,..\, v_{je}]$, $v_{jb} < v_{je}$, INV_i includes INV_j if $v_{ib} \le v_{jb}$ and $v_{je} \le v_{ie}$. Interval $INV_i <_{[]} INV_j$ if $v_{ie} < v_{jb}$, $INV_i >_{[]} INV_j$ if $v_{ib} > v_{je}$, and $INV_i =_{[]} INV_j$ if $v_{ib} = v_{jb}$ and $v_{ie} = v_{je}$. INV_i and INV_j are disjoint if either is less ($<_{[]}$) than the other, and overlap unless they are disjoint.

For any logical time t, $pulse(t)$ denotes the *pulse* component of t. The *var*, *pId*, and *rank* components are denoted similarly. Also, $prefix(t)$ denotes the first two components of t, i.e., $prefix(t) = (pulse(t), var(t))$. Similarly, $suffix(t) = (pId(t), rank(t))$.

The term *operation* refers to both original and composite operations. For any operation OP_i, $pulse_{[]}(OP_i) = [pulse(t_{ib}) \,..\, pulse(t_{ie})]$. The *ts* (timestamp), *var*, *pId*, *rank*, *prefix*, and *suffix* intervals are defined and denoted similarly. For example, $ts_{[]}(OP_i) = [t_{ib} \,..\, t_{ie}]$ and $prefix_{[]}(OP_i) = [prefix(t_{ib}) \,..\, prefix(t_{ie})] = [(pulse(t_{ib}), var(t_{ib})) \,..\, (pulse(t_{ie}), var(t_{ie}))]$. Figure 4 illustrates the intervals defined over an operation's timestamp. We say OP_i and OP_j are disjoint if their timestamp intervals are disjoint. Similarly $OP_i <_{[]} OP_j$ if $ts_{[]}(OP_i) <_{[]} ts_{[]}(OP_j)$ and $OP_i >_{[]} OP_j$ if $ts_{[]}(OP_i) >_{[]} ts_{[]}(OP_j)$.

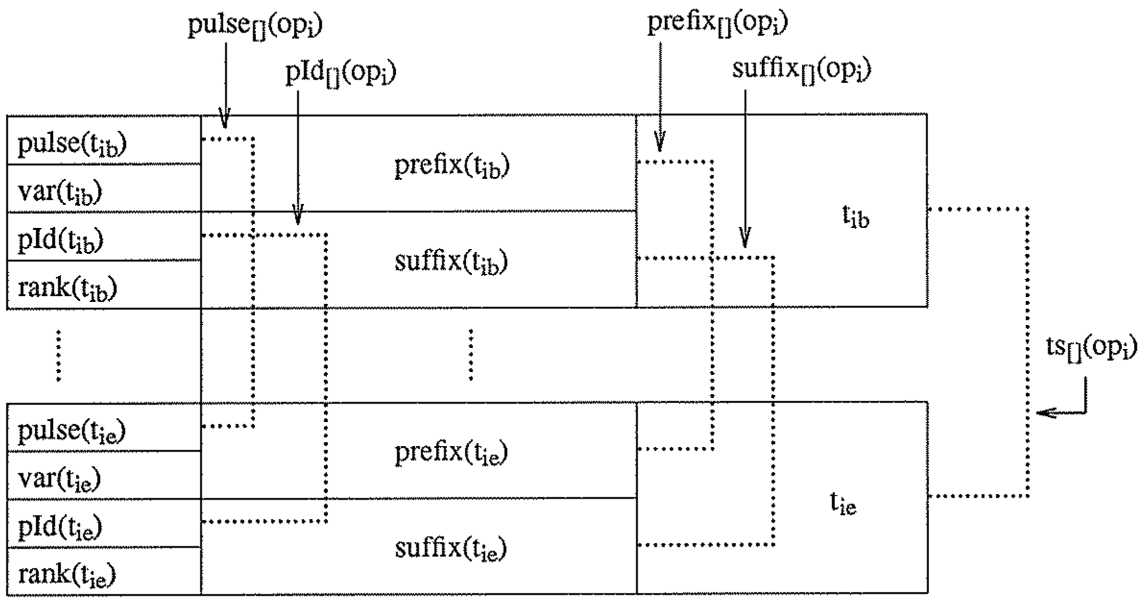

Figure 4. Intervals for Operation OP_i with Timestamp (t_{ib}, t_{ie}).

Definition. For any operation OP_i, let $\overline{OP_i}$ denote an operation that represents OP_i, and let $\hat{OP}_i$ denote the operation representing OP_i that is executed at the MM. Recall an operation represents itself. If OP_i is received by the MM without being (further) combined, $OP_i = \hat{OP}_i$.

Definition. For any operation OP_i, the *target execution time* (or timestamp) for OP_i, denoted $t_{target}(OP_i)$, is the execution timestamp OP_i would have in the absence of combining, i.e., $t_{emit}(OP_i) + DIST$, where $DIST$ is as defined before. On a non-combining isotach network, the actual and the target execution timestamps are the same.

Definition. The *recursive expansion* of OP_i is the sequence of original operations represented by OP_i in the order in which they are executed in the serial execution equivalent to OP_i.

Definition. For any two operations, OP_i and OP_j, on the same variable, OP_i *effectively precedes* OP_j, if 1) $\hat{OP}_i \neq \hat{OP}_j$ and $\hat{OP}_i$ is executed before $\hat{OP}_j$ or 2) $\hat{OP}_i = \hat{OP}_j$ and OP_i precedes OP_j in the recursive expansion of $\hat{OP}_i$.

Definition. For any two switches, S_i, and S_j, S_i is the *immediate predecessor* of S_j if there is a direct link from S_i to S_j and operations are routed over that link. If S_i is the immediate predecessor of S_j, then S_j is the *immediate successor* of S_i.

We list several basic properties of timestamps and the pId-intervals of switches that follow directly from the convexity of the network and the rules for combining operations, in particular the rule that a switch can combine operations only in the proper orientation and only if these operations access the same variable and arrive at the switch in the same pulse:

P1. The pId-intervals of all immediate predecessors of any switch are disjoint.

P2. For any operation OP_i, $prefix(t_{ib}) = prefix(t_{ie})$ and $t_{ib} \leq t_{ie}$.

P3. For any variable V, any switch S, and operation OP_i on v that goes through S, the pId-interval of S for V includes $pId_{[]}(OP_i)$.

P4. For any operation OP_i, $suffix_{[]}(\hat{OP}_i)$ includes $suffix_{[]}(\overline{OP_i})$ includes $suffix_{[]}(OP_i)$.

P5. For any operation OP_i, $prefix_{[]}(t_{receive}(\hat{OP}_i)) = prefix_{[]}(t_{target}(OP_i))$.

5.4. Proof

We show combining operations in an isotach network preserves the effective execution order of operations. Therefore, combining is done invisibly and does not affect the semantics of isochrons.

Lemma 1. *Any two operations, OP_i and OP_j, arriving at any switch S are disjoint.*

Proof. Either $prefix(t_{ib}) \neq prefix(t_{jb})$ (case 1) or $prefix(t_{ib}) = prefix(t_{jb})$ (case 2).

Case 1. Since $prefix(t_{ib}) = prefix(t_{ie})$ and $prefix(t_{jb}) = prefix(t_{je})$ (by P2), the prefix-intervals of OP_i and OP_j are disjoint. Therefore OP_i and OP_j are disjoint.

Case 2. The prefix-intervals of OP_i and OP_j are the same (by P2). Therefore OP_i and OP_j access the same variable v and arrive at S in the same pulse. Either they arrive on different inputs (case 2a) or on the same input (case 2b).

Case 2a. Since the pId-intervals for v of S's immediate predecessors are disjoint (P1) and the pId-interval for v of a switch includes the pId-interval of any operation on v emitted by the switch (P3), the pId-intervals of OP_i and OP_j are disjoint. Since their prefix-intervals are the same and their pId-intervals are disjoint, OP_i and OP_j are disjoint.

Case 2b. Since OP_i and OP_j arrive at S on the same input, they are emitted by the same immediate predecessor of S. Since they have the same prefix-interval, they access the same variable V. We show OP_i and OP_j are disjoint by proving any two operations on V emitted by the same node S' in V's routing tree are disjoint. The proof is by induction on the height of S' in V's routing tree, where the height of S' is 0 if S' is a leaf node and is 1 plus the height of its highest immediate predecessor otherwise.
Inductive Hypothesis: Any two operations, OP_i and OP_j, on the same shared variable V emitted by the same node S' are disjoint.
Basis: The hypothesis is true for S' at height 0 because a node at height 0 is a PE-SIU and all operations emitted by any PE-SIU are disjoint.
Inductive Step: Assuming the hypothesis is true for S' at height $\leq k$, it is true for S' at height k+1. For any node S' at height k+1 all operations arriving at S' on different inputs are disjoint by case 2a and all operations arriving on the same input are emitted by nodes at height k or less and are disjoint by the inductive hypothesis. Thus all operations received at S' are disjoint. Every operation emitted by S' is received by S' except those operations created by combining operations received at S'. Thus all operations emitted by S' are disjoint, unless S' combines disjoint operations to create an operation that overlaps another operation it emits. By the combining algorithm, if OP_i and OP_j are combined by S', there is no operation OP_x received by S' such that $OP_i <_{[]} OP_x <_{[]} OP_j$. Therefore no combining by S' can create an operation that overlaps any other operation it emits and all operations emitted by S' are disjoint. □

LEMMA 2. *For any variable* V*, all operations on* V *are received at the MM containing* V *in timestamp order.*

PROOF. We show each MM-SIU receives operation in timestamp order by an inductive argument similar to that in lemma 1.
Inductive Hypothesis: For any node S in V's routing tree and any pair of operations OP_i and OP_j on V that both go through S, S emits OP_i and OP_j in timestamp order.
Basis: The hypothesis is true for S at height 0 because a node at height 0 is a PE-SIU and each PE-SIU emits all operations on V in timestamp order.
Inductive Step: Assuming the hypothesis is true for S at height $\leq k$, it is true for S at height k+1, where height is as defined in lemma 1. Since each input on which S receives operations on V is the output of a node in V's routing tree at height k or less and since each channel is FIFO all operations on V arrive on each input in timestamp order. In a non-combining isotach network, the switch merges the streams of operations arriving on its inputs by waiting until there is a token or operation on each input and choosing the operation with the lowest timestamp for routing. Thus in the absense of combining, S emits operations on V in timestamp order. In a combining isotach network, the timestamps are intervals of logical time. If S does not combine OP_i and OP_j then S routes OP_i before OP_j if $t_{ib} < t_{jb}$. Since all operations received at a switch are disjoint (lemma 1), $t_{ib} < t_{jb} \Rightarrow OP_i <_{[]} OP_j$. If S combines OP_i and OP_j, then for any other operation OP_x that goes through S, OP_x is either less than OP_{ij} and S emits OP_x before OP_{ij} or OP_x is greater than OP_{ij} and S emits OP_x after OP_{ij}. Therefore S emits operations on V in timestamp order.

LEMMA 3. *For any two operations,* OP_i *and* OP_j*, on the same shared variable* V*,* $\hat{OP}_i \neq \hat{OP}_j$ *and* $t_{target}(OP_i) <_{[]} t_{target}(OP_j) \Rightarrow t_{receive}(\hat{OP}_i) <_{[]} t_{receive}(\hat{OP}_j)$. In other words, if OP_i and OP_j are not combined into the same operation and if OP_i would, in the absense of combining, have an earlier execution timestamp than OP_j, then the operation representing OP_i is executed before the operation representing OP_j.

PROOF. Either $prefix_{[]}(t_{target}(OP_i)) <_{[]} prefix_{[]}(t_{target}(OP_j))$ (case 1) or $prefix_{[]}(t_{target}(OP_i)) =_{[]} prefix_{[]}(t_{target}(OP_j))$ (case 2).

Case 1. By P5, $prefix_{[]}(t_{target}(OP_i)) <_{[]} prefix_{[]}(t_{target}(OP_j)) \Rightarrow prefix_{[]}(t_{receive}(\hat{OP}_i)) <_{[]} prefix_{[]}(t_{receive}(\hat{OP}_j))$. Therefore $t_{receive}(\hat{OP}_i) <_{[]} t_{receive}(\hat{OP}_j)$.

Case 2. Since the prefix-intervals of $t_{target}(OP_i)$, $t_{receive}(\hat{OP}_i)$, $t_{target}(OP_j)$, and $t_{receive}(\hat{OP}_j)$ are all the same (P5), $t_{receive}(\hat{OP}_i) <_{[]} t_{receive}(\hat{OP}_j)$ if $suffix_{[]}(t_{receive}(\hat{OP}_i)) <_{[]} suffix_{[]}(t_{receive}(\hat{OP}_j))$. Since $t_{receive}(\hat{OP}_i)$ and $t_{receive}(\hat{OP}_j)$ are disjoint (lemma 1) and their prefix-intervals are the same, their suffix-intervals are disjoint.

By P4, $suffix_{\square}(t_{receive}\ (\hat{OP}_i))$ includes $suffix_{\square}(t_{target}\ (OP_i))$ and $suffix_{\square}(t_{receive}\ (\hat{OP}_j))$ includes $suffix_{\square}(t_{target}\ (OP_j))$. Since $t_{target}(OP_i) <_{\square} t_{target}(OP_j)$ and the prefix-intervals of $t_{target}(OP_i)$ $t_{target}(OP_j)$ are the same $suffix_{\square}(t_{target}(OP_i)) <_{\square} suffix_{\square}(t_{target}(OP_j))$. Therefore $suffix_{\square}(t_{receive}(\hat{OP}_i)) <_{\square} suffix_{\square}(t_{receive}(\hat{OP}_j))$. Therefore $t_{receive}(\hat{OP}_i) <_{\square} t_{receive}(\hat{OP}_j)$. □

LEMMA 4. *For any two operations, OP_i and OP_j, on the same shared variable* V, $\hat{OP}_i = \hat{OP}_j$ *and* $t_{target}(OP_i) <_{\square} t_{target}(OP_j) \Rightarrow OP_i$ *precedes* OP_j *in the recursive expansion of* $\hat{OP}_i$. In other words, if OP_i and OP_j are combined into the same operation, $\hat{OP}_i$, and, in the absence of combining, OP_i would have an earlier execution timestamp than OP_j, then execution of $\hat{OP}_i$ is equivalent to a serial execution of some sequence of operations in which OP_i comes before (not necessarily immediately before) OP_j.

PROOF. Since $\hat{OP}_i = \hat{OP}_j$, there is some switch S that combines an operation $\overline{OP_i}$ representing OP_i and an operation $\overline{OP_j}$ representing OP_j. Since they access the same variable and arrive in the same pulse, $prefix_{\square}(\overline{OP_i}) =_{\square} prefix_{\square}(\overline{OP_j})$. By the argument used in case 2 of lemma 3, $suffix_{\square}(\overline{OP_i}) <_{\square} suffix_{\square}(\overline{OP_j})$. Therefore, $\overline{OP_i} <_{\square} \overline{OP_j}$ and S combines these operations to create OP_c where $f_c = f_j \circ f_i$, i.e., OP_c is equivalent to execution of $\overline{OP_i}$ before $\overline{OP_j}$. Later switches can further combine OP_{ij} but cannot change the orientation of OP_i and OP_j within it. Since f_j precedes f_i in the composition of functions specified by $\hat{OP}_i$, OP_i precedes OP_j in the recursive expansion of $\hat{OP}_i$ [KRS88]. Therefore execution of $\hat{OP}_i$ is equivalent to a serial execution in which OP_i is executed before OP_j. □

THEOREM *Operations can be combined in an isotach network consistently with the semantics of isochrons.*

PROOF. Two executions of the same program are equivalent if each shared variable is accessed by the same operations in the same order in both executions. By lemmas 3 and 4, for any two operations, OP_i and OP_j, on the same variable, OP_i effectively precedes OP_j in an execution on an isotach combining network iff OP_i effectively precedes OP_j on a non-combining isotach network. Therefore, for each execution E_c on an isotach combining network there is an equivalent execution E_s on a non-combining isotach network. Execution E_s is the execution in which the same operations are emitted at the same logical times as in E_c. □

6. CONCLUSION

This paper shows that operations from different atomic actions can be combined in a recombining network. The result is limited to *flat* atomic actions, atomic actions in which all the operations on shared variables can be issued as a batch. In another paper [WiR89], we show how to execute other types of atomic actions using isochrons by splitting each access to a shared variable into a scheduling and an assignment step. Our current work includes investigating the combinability of split operations as a way to extend the class of atomic actions that can be combined within a recombining network.

ACKNOWLEDGEMENTS. This work is supported in part by DARPA, NASA, and the University of Maryland and in part by JPL Contract #957721. The authors acknowledge the help of Ray R. Wagner, Jr., in naming *local synchrony* and in contributing a counterexample leading to the convexity requirement.

REFERENCES

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

[BGS89] Y. Birk, P. B. Gibbons, J. L. C. Sanz and D. Soroker, A Simple Mechanism for Efficient Barrier Synchronization in MIMD Machines, Tech. Rep. RJ 7078 (67141), IBM, October 1989.

[DuS90] M. Dubois and C. Scheurich, Memory Access Dependencies in Shared-Memory Multiprocessors, *IEEE Trans. on Software Eng. 16*,6 (June 1990), 660-673.

[Fid91] C. Fidge, Logical Time in Distributed Computing Systems, *Computer*, August 1991, 28-33.

[GGK83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer --- Designing an MIMD Shared Memory Parallel Computer, *IEEE Transactions on Computers 32*,2 (February 1983), 175-189.

[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.

[Lam78] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, *Comm. ACM 21*,7 (July 1978), 558-565.

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

[Mat88] F. Mattern, Virtual Time and Global States of Distributed Systems, *Parallel and Distributed Algorithms*, 1988, 215-226.

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

[PfN85] G. F. Pfister and V. A. Norton, Hot Spot Contention and Combining in Multistage Interconnection Networks, *IEEE Transactions on Computers 34*,10 (October, 1985), 943-948.

[Pfi85] G. F. Pfister, et al., The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, *Int. Conf. on Parallel Processing*, 1985, 764-771.

[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.

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

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

[RSL78] D. Rosenkrantz, R. Stearns and P. Lewis, System-level Concurrency Control for Distributed Data Bases, *ACM Transaction on Database Systems 3*,2 (1978), 178-98.

[TuR88] L. W. Tucker and G. G. Robertson, Architecture and Applications of the Connection Machine, *Computer 21*,8 (August 1988), 26-38.

[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.

[WiR90] C. Williams and P. F. Reynolds, Jr., Delta-Cache Protocols: A New Class of Cache Coherence Protocols, Tech. Rep. 90-34, University of Virginia, Department of Computer Science, December, 1990.

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