A Local Synchrony Implementation: Banyan Networks

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

Computer Science Report No. TR-91-38
December 1991

A Local Synchrony Implementation: Banyon Networks

Abstract

Local Synchrony is a synchronization method for ensuring correct temporal relations among accesses to a virtual shared memory. Local Synchrony is a conservative timestamp ordering protocol. It implements a logical time system in an asynchronous architecture.

We describe an implementation of Local Synchrony on a Banyan equidistant network architecture consisting of 2-by-2 switching elements. Our description includes the timestamp ordering system, the transport and routing mechanisms, and access flow control. Local Synchrony also facilitates the implementation of FIFO combining [Ran87].

Because Local Synchrony is a conservative system, its operation differs from the norm for asynchronous architectures. The question of deadlock freedom becomes important, as blocking is possible in a switch. We present proof, using special information distributing messages called ghost messages, that Local Synchrony is deadlock free for this class of networks.

Also, we present an analytical model of our implementation which is similar to Jenq's [Jen83]. This model allows us to predict costs for Local Synchrony in comparison with an accepted analytical model.

1.0 Introduction

We propose an implementation for Local Synchrony on a simple Banyan network. Local Synchrony is a synchronization method for virtual shared memory parallel architectures using logical time. In a virtual shared memory architecture, the view of any given process, separate from its own private, local storage, is of a shared memory space, regardless of the actual implementation of memory. Local Synchrony is a conservative system which implements a logical time system in an asynchronous architecture.

The motivation behind our research is the support of *isochrons* [Wil90], a concept derived from parallel operations [Wag87] [RWW89]. An isochron consists of a set of memory accesses emitted by a given process which appear to be executed simultaneously, so that memory coherency is maintained. Preserving coherency of memory means that it must not be possible for a process to receive an inconsistent state representation due to interleaving of other accesses with those of the given process.

Isochrons are a limited form of *atomic action*. The general form of atomic actions is a group of operations which are performed *atomically* [OwL82], meaning that no process other than the issuer can observe or affect an intermediate state in the execution. In a system implementing isochrons, no process, *including the issuer*, can observe or affect any intermediate state in the execution of an isochron. Thus, isochrons do not support global data-dependent operations.

In Local Synchrony, an isochron takes the form of a set of accesses issued from a given process during a given logical time pulse. Local Synchrony assigns a unique logical timestamp to each access and guarantees that accesses are executed on a given shared memory location in timestamp order.

Local Synchrony may be cast as a conservative timestamp ordering system. Local Synchrony is constrained to produce sequentially consistent serial schedules for shared memory accesses. The atomicity of isochrons is guaranteed with no possibility of rollback or aborted accesses. In the database literature, Bernstein and Goodman [BeG81] state that, in a conservative system, "When a scheduler receives an operation O that might cause a future restart [logical time conflict], the scheduler delays O until it is sure that no future restarts [logical time conflicts] are possible." The description and implementation of Local Synchrony fits this characterization.

Conservative timestamp ordering systems have been proposed in the database literature [BeG80] [Mil79]. These types of systems are generally based on global knowledge about accesses currently in the system and the current timestamps of processes. Local Synchrony requires only local processing decisions for implementation.

Reed [Ree83] suggests a timestamp ordering implementation of atomic actions based on timestamping by physical clocks and the use of version histories in shared memory. A physical clock is located at each processing node, and the clocks are kept loosely synchronized by a global clock synchronization algorithm. In this implementation the problem of aborted actions/rollback still exists.

Concurrency control as applied to parallel architectures and operating systems has been studied for the last fifteen years. In [Lam78], Lamport proposes a system of *logical clocks*, which would provide partial ordering to the events of a system of processes. Lamport uses the guaranteed sequential consistency of these orderings to solve synchronization problems.

Jefferson [Jef83] [Jef85] has proposed a more general theory of logical time in *Virtual Time Systems*. The goal of Virtual Time Systems is to create a temporal system in which a distributed computation can be embedded which will provide correct, logical orderings of the events of that computation. Again, the correctness of the logical orderings corresponds to their providing a proper serial schedule. Jefferson's protocol, designed for use in parallel simulation, is optimistic, utilizing state-saving and rollback in order to meet its clock conditions.

Local Synchrony is a conservative timestamp ordering protocol for virtual shared memory accesses in parallel architectures. Local Synchrony guarantees serial scheduling and sequential consistency, as well as the atomicity of isochrons, without the possibility of aborted accesses or rollback. Local Synchrony may be implemented with word or variable length accesses and is highly compatible with FIFO combining [Ran87]. We describe an implementation of Local Synchrony on a Banyan equidistant network architecture consisting of 2-by-2 switching elements. Our description includes discussion of access size, transport mechanism, and buffer sizes and allocation.

Because Local Synchrony is a conservative system, its operation differs from the norm for an asynchronous architecture. The question of deadlock freedom becomes important, as blocking is possible in a switch. We present proof, using special information distributing messages called ghost messages, that Local Synchrony is deadlock free in this architecture.

In [Jen83], Jenq presents an analytical model for the performance of Banyan networks in a synchronous, single-buffered, packet-switching system. Jenq's model is based on the iterative steady-state solution of probabilistic equations governing the operation of the network, and provides measures for network throughput and packet delay. We present a similar model for the Local Synchrony implementation discussed here.

2.0 Local Synchrony

A Local Synchrony implementation is a conservative (logical timestamp) ordering system for shared memory accesses in parallel systems. Local Synchrony maintains and guarantees a total ordering for the execution of shared memory accesses. Locally Synchronous entities in an implementation can include processing elements (PEs), memory modules (MMs) and switching elements.

As stated in section 1.0, Local Synchrony may be cast as a conservative timestamp ordering system. Local Synchrony assigns a logical (rather than physical) timestamp to each shared memory access. An entity operating in a conservative manner is constrained so as not to perform any timestamped action until it has, with certainty, performed all actions with earlier timestamps. For example, a memory module (MM) operating conservatively cannot execute an access with timestamp t until it is guaranteed that all accesses to that MM with timestamps less than t have been executed. Local Synchrony requires only that MMs operate conservatively in processing shared memory accesses. However, the implementation of Local Synchrony discussed in this report requires conservative operation by switches and PEs as well.

Local Synchrony may be characterized by the serializability, or constructible serial schedule, of all shared memory accesses performed. This serial schedule is determined by the timestamp ordering of shared memory accesses. Each entity (PE, MM, or switch) stays consistent with this serial schedule by loose synchronization with neighboring nodes. Note that this is local, rather than global synchronization.

This loose synchronization between entities is implemented in the following way: each entity in the system periodically emits a set of control signals, or *tokens*, one along each of its output connections to other nodes. These tokens represent the boundaries between the logical timesteps of the entity's own logical clock. For the purposes of further discussion, we will define a pulse to be that period of time between the sending of two sequential tokens from the same entity. During a pulse, a PE may emit zero or more accesses (accesses may only originate in PEs).

In order to implement *local* synchronization, an entity must route all accesses received from its neighbors during the previous pulse, before generating its next pulse. In the case of the Banyan network, routing entities are switches. Accesses may be routed without regard to their timestamp order within the current pulse, in which case accesses must be buffered and sorted at the MM, or by merging sorted streams of accesses by timestamp. The implementation reported here requires that accesses enter the network in sorted streams (by timestamp), and that network switches merge the sorted streams of accesses. Our choice of this protocol is based on scalability factors pertaining to the amount of buffer space necessary for implementation.

The logical time of execution of each access is determined by the issuing PE through explicit timestamping, implicit timestamping (determined by ordering conventions implemented within the system), or a combination of the two. The timestamp for a given access is an ordered pair (*tick, tock*) [Wil90], where the *tick* component is itself an ordered pair (*pulse, pid*). The *pulse* component of the access is the pulse in which the access will be executed.

The *pid* component is an id unique to the issuing PE. The use of this *pid* in the timestamp assigns a disjoint interval of logical time during each pulse to the accesses issued by the given PE. This disjoint interval of logical time provides the basis for isochrons [Wil90], parallel memory access operations. Isochrons provide atomic access to multiple shared variables without locking. Local Synchrony was developed to facilitate the use of isochrons.

The *tock* component of the timestamp provides sequentially consistent ordering for any accesses to the same shared variable, from the same process, in the same pulse. When implementing Local Synchrony on a Banyan network, this component is implicit in the design of the architecture, which guarantees that accesses emitted from the same PE cannot change their order during transmission through the network.

In [RWW89], the notion of *convexity* was introduced. For a wide class of networks, including Banyan networks, there exists a convex labeling, i.e. an assignment of *pids* to PEs such that the source of each input channel to a given switch is a set of PEs with contiguous pids. For example, consider a switching element having two input channels, A and B. If the inputs to channel A can come only from PEs with pid 0, 1, 2, and 3, and the inputs to channel B can come only from PEs with pid 4, 5, 6, and 7, then the inputs to the switching element are convex. One can see that its output channels can transmit only accesses from a contiguous set of pids (0-7). If the property holds for every element in the switching network, then the network is convex.

The implementation of Local Synchrony in convex systems is highly compatible with access combining [GLR81]. Using access protocols described in [KRS88], accesses may be combined in the network in such a way that the effect of the combined access when executed at shared memory is identical to the effect of any given ordering of the original accesses. Since ordering in combining is unimportant, the ordering necessary for Locally Synchronous operation may be preserved if the system is convex.

In order to facilitate combining of accesses PEs can add a fourth component to the timestamp (although this component is for combining purposes only and does not affect the logical ordering of accesses). Accesses are sorted and issued in timestamp order, by pulse, combining component, pid, and rank (of accesses from the same PE to the same shared memory location). The combining component of the timestamp is generally implemented as the address of the memory location to be accessed, although several options are possible.

Switches in this implementation perform a sorted merge, which brings accesses bound for the same variable together for combining. Accesses are combined in the orientation (serial ordering) which preserves the pid ordering that is essential to correct ordering of access execution.

We note here that our discussion assumes that accesses may be fully pipelined. This is not the case in the general operation of parallel architectures. In order to maintain sequential consistency, pipelining is only allowed in 'safe' situations, as indicated by algorithms which detect data dependencies. In [ShS88], Shasha and Snir point out that "Pipelining of memory requests is required in order to mask the (relatively) large latency of the interconnection network." Local Synchrony guarantees implicitly the sequential consistency of shared memory accesses, and thus pipelining becomes the norm rather than the exception.

We have here presented a general overview of Local Synchrony. More in-depth discussion may be found in [Wil90], and [RWW89].

3.0 Banyan Implementation Plan

We consider the implementation of Local Synchrony on an Ultracomputer-like Banyan network, consisting of $\log_2 n$ stages of $n/2$ switching elements per stage, where n is the number of processors (PEs) and memory modules (MMs). For this implementation, switching elements are assumed to be 2-input, 2-output. An $n=16$ Banyan architecture is illustrated in figure 1. The basic, indivisible shared memory access for such an architecture is assumed to be the atomic *fetch-and-op*.

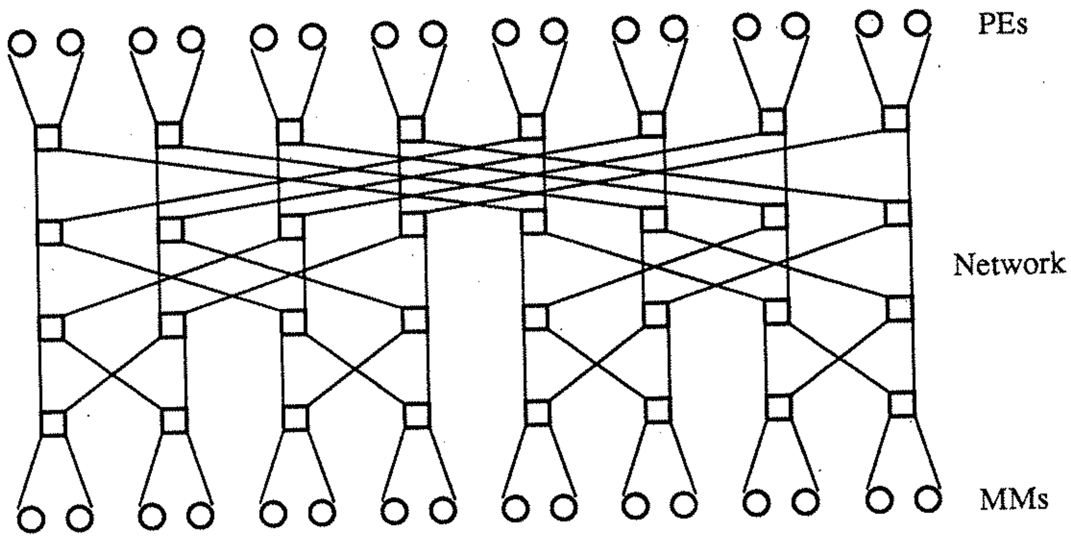

FIGURE 1. An n=16 Banyan architecture.

We identify seven issues of importance in any such implementation with respect to switch architecture: the transport mechanism, the routing mechanism, buffer management, flow control, the message ordering system, the combining system, and the deadlock avoidance system. The first four of these are identified in [ReF87]. A diagram of a simple SE is shown in figure 2.

3.1 Message Ordering System

We discuss this issue first because several of the other issues rely on the decisions made here. Local Synchrony requires that shared memory accesses each be assigned a unique timestamp which represents a logical execution time for that access. Accesses are then ordered by timestamp at some point prior to execution. This timestamp includes information pertaining to the logical time of execution, the ID of the sending PE, and the access order for the accesses from that PE. Combining information (the destination address) is also included in the timestamp if the implementation is to include FIFO combining.

Accesses could be transferred through the network unsorted, leaving the MMs to ascertain when it is possible to conservatively execute a given access. Scalability arguments lead us to choose a system in which the sending PEs sort accesses before emission into the network, which acts as a merging network for sorted streams of accesses. Note that a given PE must generate *all* accesses to be emitted during a given logical time pulse before any can be actually emitted, unless the PE can guarantee that accesses will be generated in timestamp order.

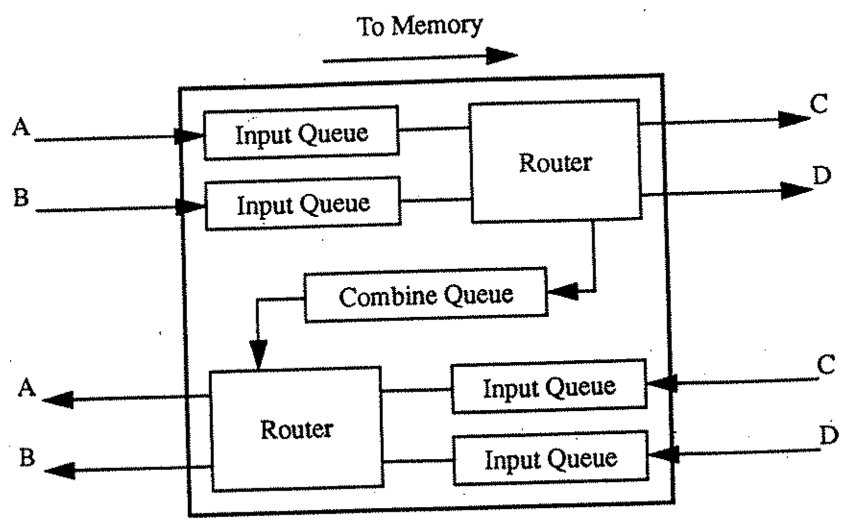

FIGURE 2. Simple SE implementation.

Local Synchrony only requires that accesses be executed in timestamp order at shared memory. This means it is possible to implement a separate, low-latency network to return the results of executed accesses to the PEs. Such a network would route accesses without regard to their timestamps, which could shorten access latency times. However, such an implementation eliminates the ability of the system to perform FIFO combining, as discussed in section 6. In our further discussion, we assume that accesses are kept sorted on return from shared memory, in order to show the feasibility of FIFO combining.

3.2 Transport Mechanism

We choose the store-and-forward access transport paradigm for the implementation of Local Synchrony.We assume that shared memory accesses have a fixed size, which greatly simplifies the model (for this discussion, we assume word access, although Local Synchrony is not limited in this respect). Because the implementation of Local Synchrony provides a basis for isochrons, word size accesses may be used without losing the flexibility of variable size memory access. An isochron provides atomic access to any number of shared memory words.

The size of the shared memory access packet depends on several factors. Generally, however, the additional information required by a Local Synchrony implementation can be implicitly represented. Variable factors in the packet size include the number of PEs and size of the virtual shared memory, the size of operands for the memory operations to be performed, and whether combining is in effect. FIFO combining, as discussed in section

7.0, can add to the size of the packet, if packets keep information pertaining to where they have been combined.

For example, consider an Ultracomputer architecture with 64K processors, an address space of four gigabytes, and 32-bit operands. A packet (data unit) size of 16 bytes would be sufficient to implement Local Synchrony in such a system. Four bytes would cover the destination address in shared memory (2^{16} address spaces per MM). The PID element of the timestamp would consume 2 bytes. Other elements of the timestamp can be implicitly represented (we will assume that separate 'dummy' packets mark changes in the logical time pulse, and that accesses from the same PE cannot 'pass' one another.). The operation to be performed, along with a ghost message bit, would fit easily into one byte. The operand of the operation fills a four byte word. Finally, combining information for the network (16 stages) would take two bytes. This leaves more than three bytes of space for error correcting codes and other information.

Buffering complexity using the store-and-forward paradigm with static sized packets is relatively simple. A minimum of one packet buffer is necessary at each input channel at all times. More complex buffering strategies are possible as long as they meet this criterion.

3.3 Routing Mechanism

The routing mechanism employed by a Local Synchrony implementation may be separated into two distinct steps. First, the router chooses the next packet to be routed from among those at the head of each of its input channels. This decision is made by choosing the lowest logical timestamp. Note that because the streams of input packets are already sorted, this choice amounts to a conservative choice of the earliest timestamped not-yet-routed packet. It is further important to note that the routing mechanism must have a packet (or a ghost message, which in this implementation is represented by a packet) on each of its input channels to make a decision as to which is next to be routed.

The second step in the routing process is to route the chosen packet to the correct output channel. If we assume that the portion of shared memory held by a given MM is a contiguous set of addresses, and that the MMs are ordered contiguously in the architecture, then this decision amounts to checking whether the destination address of the packet is greater than or less than a given address, and choosing the output channel accordingly.

3.4 Deadlock Avoidance System

To avoid deadlock within the network, it is necessary to disseminate information from SE to SE as soon as it is available. This is done with the use of *ghost messages* [Ran87] [RBJ88]. Ghost messages are similar to null messages [ChM79] from the parallel simulation literature. Null messages are used to insure deadlock freedom in groups of processes which communicate via message passing in a parallel simulation.

When an SE passes (or is waiting to pass) an access on a given output channel, it's other output channel may be idle. This condition can cause deadlock. Deadlock can be avoided by passing a ghost message, a copy of the temporal components of the access most recently sent (or waiting to be sent) on the other output channel. The timestamp of this ghost message can be used to break potential deadlocks further along in the network.

Ghost messages are identified by a special bit set within the access packet. Alternatively, a ghost message may be recognized by the fact that its destination address will never be within the range of those reachable from the receiving SE. Ghost messages require little or no processing and do not clog the network, as they utilize otherwise idle communication channels and otherwise empty buffers.

The information which an SE receives from a ghost message is the lowest possible timestamp of any future access which will arrive on that channel. This information allows the SE to make further routing decisions and break possible deadlocks. An incoming ghost message which becomes the next to be routed by a given SE must be propagated along each of that SEs output channels.

Note that the information contained in a ghost message is superseded by the arrival of a normal access or another ghost message, each of which would have a later timestamp. Thus, ghost messages are overwritten (erased) in such instances. The erasure of ghost messages due to being superseded by new ghost messages or actual accesses serves to limit the proliferation of ghost messages in the network, although it is possible for ghost messages to reach the edge of the network, at which time they would be deleted.

A proof of deadlock freedom based on ghost messages is presented in section 4.0.

3.5 Buffer Management

Buffer management can be simple or complex in this implementation. Simple buffer control reserves some number of packet buffers for each input channel. More complex buffer management can draw from a pool of buffers to queue incoming and outgoing packets on all channels. Local Synchrony requires that each input channel have access to at least one packet buffer (or already be using one) at all times to avoid buffer deadlock. An implementation might also include buffers on output for flow control purposes. Output buffers can be considered to be extra input buffers for the next stage for the purposes of analysis. Note that the number of buffer spaces necessary to implement Local Synchrony in this architecture is directly proportional to the number of communication channels.

Because Local Synchrony is a conservative system, the simple buffer management scheme would appear to be warranted. Care must be taken when deciding how many buffers to assign to a given input channel, however, as more buffers will have adverse effects on the combining system, as discussed in section 7.0.

3.6 Flow Control

The flow control of packets within the network may be accomplished with a receiver controlled packet-flow protocol. We will assume that packet transmission fault-tolerance is controlled by a lower-level send-acknowledge system which will not be discussed here.In normal operation, the receiving node uses a control line to signal the sender that it has open input buffers for that channel. When the receivers' input buffers are full, the control line signal is changed, notifying the sender that no input will be accepted until the control signal returns to normal.

Because Local Synchrony depends on the continual flow of information for the network to operate smoothly, this receiver-controlled protocol must be augmented by allowing receivers to request action from idle senders when necessary. Receivers would use another control line to request action on a given channel when holding input on other channels. This request requires the sender to make progress as soon as possible. Senders can in turn request action on their own input channels if necessary.

This system would be especially useful at the input stage of the network, where SEs could request that idle (with respect to shared memory access) PEs pass information which will allow the network to avoid blocking. This information would generally take the form of incrementing the PEs local logical clock.

A flow control policy must also be implemented to use the mechanism described. This policy would depend on the buffer management mechanism in place. In a simple system, where each input channel is permanently assigned a single packet buffer, this policy would be straight forward. Where buffers are dispensed from a pool, however, the flow control policy must enforce limits on the number of buffers available to any one channel, and guarantee that each channel has access to at least one buffer at all times. This guarantee is necessary to avoid buffer deadlock within the network.

For example, a two-input, two-output switching node might utilize a pool of eight input buffers. A flow control policy might dispense buffers to input channels with the restriction that neither of the channels may at any time utilize more than six of the available buffers.

3.7 Combining System

Dancehall type networks like the Banyan network are highly compatible with access combining [GLR81] [KRS88]. Because the implementation presented here assumes that accesses are ordered during transmission to and from shared memory, accesses combined at a given SE will return in the same order in which they left that switch. This condition allows the implementation of FIFO combining, which allows faster processing of combining information without changing the combining mechanism.

The size of a buffer in a combining information queue is the same as the size of an access, given that accesses themselves carry information regarding whether they were combined at a given stage (see section 2.0). The combining of two accesses changes, at most, only the operation and operand of the forwarded combined access, along with the combining bit for that stage of the network. The other access may be copied into the combine buffer, and used to decombine the accesses on return.

Since combined accesses return to a given SE in the same order in which they left it, decombining information can be kept in a FIFO queue [Ran87], rather than the traditional associative search queue. Each access packet can include bits which determine whether a given access was combined at a given stage in the network. The SE then need only check one bit of the returning packet and, if the packet was combined at that SE, pop the decombining information off the FIFO combining information queue.

Also, the timestamping system can be modified, by including the destination address in the timestamp. In the Banyan network, the paths to a given MM from the PEs form a tree. The paths of two combinable accesses, sent by different PEs in the same logical time pulse, will converge at some point in the network. If the streams of accesses are sorted by destination address as well as the other components of the timestamp (in this order: *pulse*, destination address, *pid*, *tock*, as defined in section 2.0), combinable accesses will be combined at this point. This guarantees that all combinable accesses within a given logical pulse will be combined, given the size limits on the combining information queues. It is expected that streamlining and simplifying the combining mechanism in these ways will expand its power and usefulness.

The size of the combining queue necessary in a given node in the network is dependent on the size of the network, the stage in which the node resides, and the amount of space allotted to communication buffers in each node. Local Synchrony causes the network beyond the outputs of a switch to act as a large FIFO queue. The size of this 'virtual' queue is the maximum number of combined accesses that a switch can have outstanding at any given time. The maximum necessary combining information queue size is given by:

$$C_{Max}=N*(I+R) \qquad \text{(EQ 1)}$$

where N is the number of stages between the node and shared memory, I is the number of input buffers per node per channel, and R is the number of return buffers per node, per channel. Note that this equation does not take into account output buffers, if they are implemented in the system. Our simple implementation does not include output buffers. For example, in a 16 stage network, with simple buffering (one input buffer per channel per node, both input and return), an SE in stage10 would require 20 combining buffers to guarantee that all combinable accesses could be combined.

It may be practical to limit the size of the combining buffer, as the probability that the network will be operating at capacity, that all inputs will be combinable, and that the 'virtual' queue ahead of a given switch will contain only accesses which passed through that switch is generally small, and more complex input buffering schemes can allocate many more buffers to a given channel, thus making C_{Max} quite large. Note that Local Synchrony is not compromised if two combinable accesses are not combined. The only effect of this situation is a potential loss of throughput.

4.0 Deadlock Freedom

Our discussion of deadlock freedom for Local Synchrony on Banyan (Equidistant) networks relies on the following assumptions about the implementation:

1. The network communicates with fault-free systems which provide input and consume output at the rate the network operates.

In other words, there is always input at the network inputs, and output is consumed immediately at network outputs. This assumption only simplifies progress assertions about the PEs and MMs attached to the network. It does not limit the applicability of the proof.

2. Switches in the network are able to buffer at least one access (packet) on each input channel, and once a packet occupies a buffer it can only make progress toward its destination; it may not be aborted.

Because Local Synchrony guarantees that accesses will ultimately be executed in order, switches must buffer incoming accesses on each input channel to make routing decisions. Accesses must not be aborted once they have entered the network, as this will, in most cases, compromise the operative conditions of Local Synchrony.

3. Switches in the network perform routing operations sequentially.

In other words, an SE may only perform one operation at a time. It may not, for example, both block trying to output a routed access while at the same time choosing and routing its next access.

Assumption 3 is a simplifying assumption which enhances the understandability of the proof. Alternatively, SEs may multiplex operations for more efficiency, given that conservatism is maintained, and that an SE will not block on a lower priority operation without returning to a previously blocked higher priority operation.

4. Switches in the network are able to make routing decisions based on a known priority scheme.

In order to implement Local Synchrony, accesses are assigned timestamps which serve to arbitrate the routing decisions of switching elements. These timestamps provide a static priority system which is guaranteed to arbitrate the routing decision for any two accesses. The routing operation of a switch is that of a merge of two sorted (by timestamp) streams of accesses.

The network itself is assumed to be constructed of two-input, two-output switching elements, and all SEs and communication channels are assumed to be fault-free.

Deadlock is characterized by the occurrence of a circular set of dependencies in which each element within the set requires progress by another element within the set before the element itself may progress. In the case of a Banyan network with the above assumptions, the set may include switching elements only.

It is easily seen that the zero-stage (one PE, one MM) and the one-stage (two PEs, two MMs, one SE) cases are deadlock free, as the largest set of SEs possible in either case has fewer than two members. We will first discuss the two-stage case (four PEs, four MMs, four SEs). We will then state lemmas leading to our proof of deadlock freedom.

Figure 3 illustrates the two-stage case. The network is shown in a). It is assumed that information flow is left to right. In b), the two possible circular dependencies are illustrated.

In each case, the left arrows (whether horizontal or diagonal) represent a dependency of consumer on producer, A stage 0 node (consumer) awaits input from a stage 1 node before it may itself progress.This is a *type 1* dependency. Since the producers (10 and 11) in this situation are also blocked, deadlock has occurred.

The right arrows (whether horizontal or diagonal) represent a dependency of producer on consumer. A node in stage 1 cannot progress because the input buffer in the node to which its next access is to be sent is full. The stage 1 node then depends on the stage 0 node to progress before it may progress. This is a *type 2* dependency.

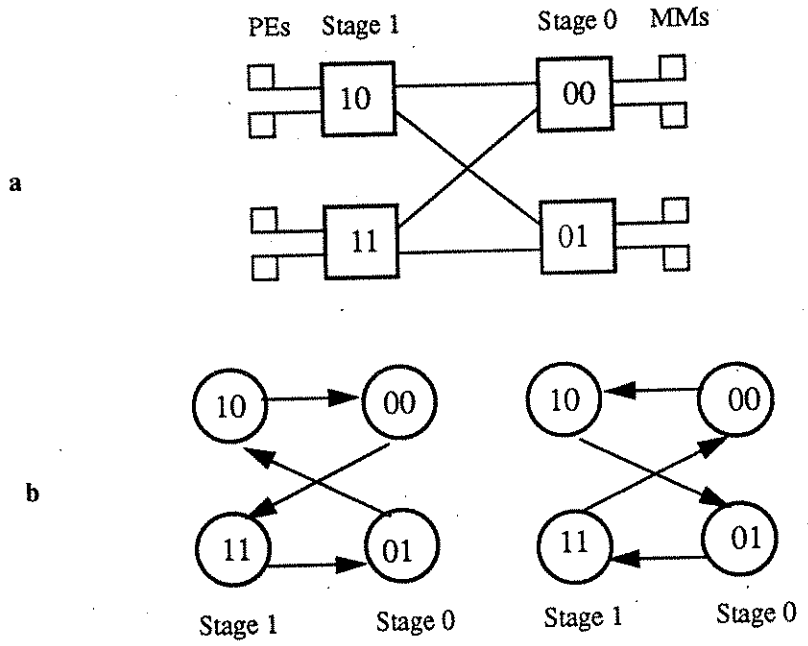

FIGURE 3. a) Two-stage Banyan Network b) deadlock dependency graphs

Deadlocks such as those shown in figure 1b can be broken if information about each producer's latest action is passed from that producer to each of its consumers. This information is already passed to the consumer that receives the latest access from the given producer. To the other consumer, this information can take the form of a *ghost message* [Ran87], which reports the last timestamp of an access sent by the producing switch.

With the information supplied by the ghost message, one of the two consumer nodes (00 and 01) will be able to progress, as it will know that no access will arrive on its empty input channel with a lower timestamp than that of the access on its other input channel. That access can thus be passed, breaking the deadlock.

For example, in figure 4a, SE 10 sends an access with timestamp t=5 to SE 00, then blocks trying to send an access with timestamp t=6 to SE 00. SE 11 sends an access with timestamp t=7 to SE 01, then blocks trying to send an access with timestamp t=8 to SE 01. Deadlock occurs. With ghost messages, however, SE 11 can send to SE 00 a ghost message based on the access sent to SE 01,having timestamp t=7, as seen in figure 2b. Now SE 00 is able to process the timestamp t=5 access sent from SE 11, as it can conservatively choose that as it's next action. Thus deadlock is avoided.

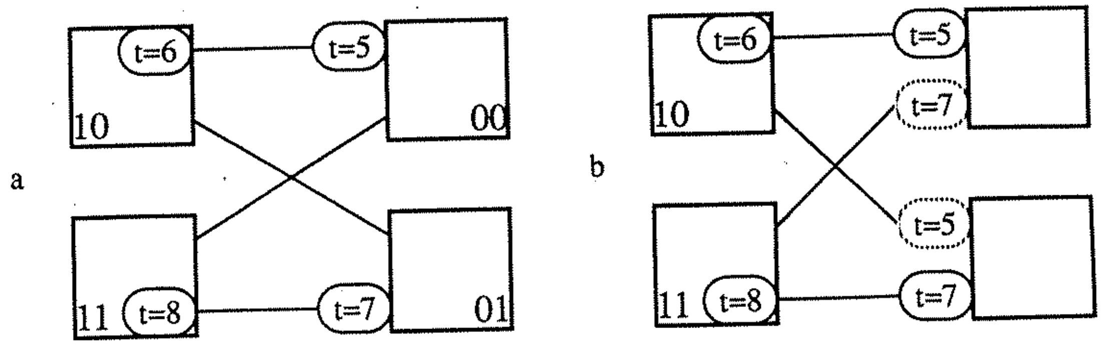

FIGURE 4. a) Deadlock b) Ghost message solution

Note that, while Local Synchrony provides a timestamping system which guarantees that all timestamps are unique, this is not a requirement for deadlock freedom.Ties may be broken, in the case of a ghost message and an access, by sending the access (and erasing the ghost message, as it is no longer useful).

Note further that a ghost message having a later timestamp can supercede an earlier ghost. message. An arriving ghost message may thus overwrite a ghost message already in the input queue. This serves to limit the proliferation of ghost messages in the network.

The ghost message may take on several forms, depending on the timestamping scheme in use. The rest of our discussion will characterize circular dependencies in larger banyan networks, and show that ghost messaging is sufficient to guarantee deadlock freedom for all such cases.

We now consider deadlock in a network of any given size. Such a situation can be characterized by a *deadlock dependency graph* (DDG) [DaS87], which includes each node in the set of deadlocked nodes, and arcs representing the dependencies of a given node upon others within the set.

We first observe that there are only two types of dependencies which can cause an SE to block and possibly contribute to deadlock in a banyan network implementing Local Synchrony. The only situations in which an SE can block are type 1) waiting on input from one of its input channels, and type 2) waiting to output on a currently full output channel.

In the figures below, type 1 blocking will be represented by a leftward arrow, and type 2 blocking by a rightward arrow. This representation corresponds to the uni-directional flow of information [in this case, from left (PEs) to right (MMs)] in the banyan network. A node's inputs are on its left, while its outputs are on its right. An example of a DDG is shown in figure 5.

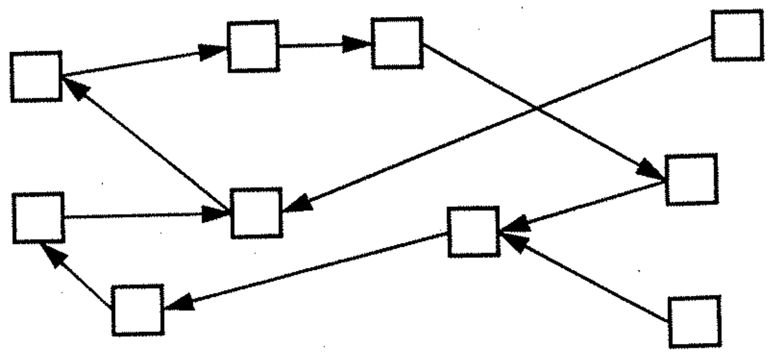

FIGURE 5. Sample deadlock dependency graph (DDG).

- Lemma 1: In a Local Synchrony implementation on a Banyan network, the DDG representation of a deadlock must include both type 1 and type 2 arcs.

Type 1 arcs are leftward pointing. Type 2 arcs are rightward pointing. By the nature of the network in question, arcs may not 'wrap around' from one side of the network to the other. Lemma 1 must hold for there to be any circularity, and thus any deadlock.

We now discuss the conditions under which ghost messages will be generated and propagated within the deadlocked set of nodes (those nodes in the DDG). Since a ghost message must use the same channels as normal accesses, only idle channels (those on which a type 1 dependency is based) may be used for their transmission among the nodes in the DDG. Thus, we will only consider, for the purposes of this proof, ghost messages originating at nodes in the DDG on which another node holds a type 1 dependency.

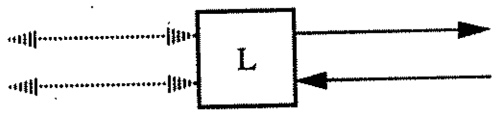

FIGURE 6. Type L node within the DDG.

Also, since a ghost message is created from information about an access to be output by the node, ghost messages are only generated at nodes in the DDG having a type 2 dependency (nodes having routed an access but unable to send it due to full buffers ahead). An example of such a *type L node* within the DDG is shown in figure 6. Note that input channel dependencies are unimportant.

- Lemma 2: At least one type L node must exist within the DDG.

A DDG must include a minimum of one circular path of dependencies which, if followed from a given node, will return to that node. Consider any circularity within a DDG. There must be at least one leftmost node within the circularity (a 'leftmost' node would be any node in the leftmost stage of all those nodes within the circularity). Any one of the set of leftmost nodes is a type L node. Neither of its inputs may be involved in the circularity, and thus both outputs must be. It cannot have the same type of dependencies (both type 1 or both type 2) on each output, or there would not be a circularity in the DDG. It must thus be a type L node. Note that any node in the DDG which is also in the leftmost stage of the network (that nearest the PEs) must, by virtue of assumption 1 above and the node's inclusion in the DDG, be a type L node.

Lemma 2 guarantees that at least one ghost message will be created in the DDG by showing that at least one node which will generate a ghost message (type L) must exist within the DDG. Lemma 3 discusses properties of ghost messages created in the DDG.

- Lemma 3: When a set of nodes is deadlocked, for each ghost message, G, generated by a node within the DDG, there exists a blocked access A, at the front of an input queue of some node in the DDG, such that timestamp t_G of G is greater than timestamp t_A of A.

Figure 7 depicts two nodes from a DDG. As discussed above, only type L nodes within the DDG will generate and propagate ghost messages. Node A is a type L node, and thus will generate a ghost message, G2, based on A2, the access it has most recently queued for output to node B. Node B, then, holds at the front of its input queue an access, A1, which, by virtue of the conservatism of Local Synchrony, is guaranteed to have timestamp $t_{A1} < t_{A2} = t_{G2}$.

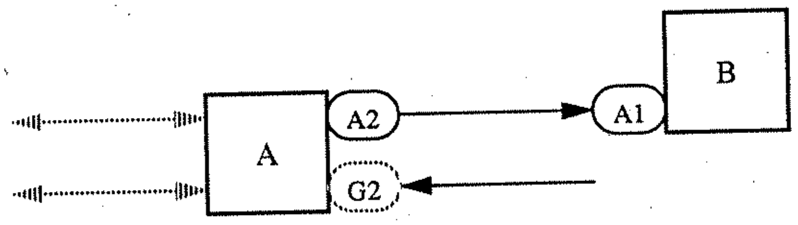

FIGURE 7. Illustration of proof of Lemma 3.

- Lemma 4: Ghost messages generated at type L nodes in the DDG will cause information (either a ghost message or an access) to be propagated on every channel in the DDG having a type 1 dependency.

Proof by contradiction: Note that type 1 dependencies must exist within the DDG, by lemma 1. Consider a channel (A), with a type 1 dependency, which will not receive information (either ghost message or access) from other nodes in the DDG. This situation is illustrated in figure 8. The node at which the dependency arrow ends (N1) cannot have a type 2 dependency on its other output channel, as it would then be a type L node, and would create a ghost message which would be propagated along channel A. Node N1 also cannot have type 2 dependencies on both its input channels, as it would thus not be blocked. N1 must then have at least one type 1 dependency on one of its input channels (B).

Channel B itself cannot have ghost message or access information propagated on it, or N1 would be able to propagate a ghost message or an access along channel A. [Note that if N1 has type 1 dependencies on *both* its input channels, then no information can be propagated on at least one of those channels (B, in our discussion), or N1 would have input on both channels and would thus be able to send information on channel A.]

The case in which no information is propagated on either input channel is a generalization of the case we discuss. Each channel is then part of a chain of dependencies which eventually reaches a contradiction. We consider one of those channels (B) here.

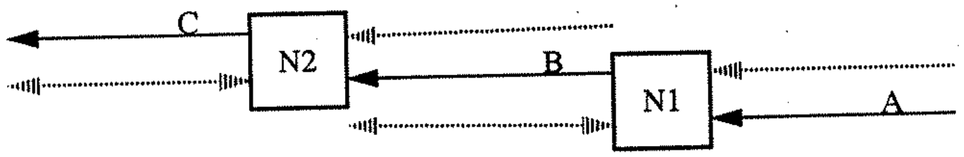

FIGURE 8. A chain of dependencies.

Since channel B also does not receive any propagated information, the node at which channel B ends (N2) must have the same dependency topology as N1, and so on. Thus a chain of dependencies exists. This chain of dependencies (channels A, B, C,...) upon which no information is propagated cannot go on indefinitely, as all dependencies are type 1, but only to the left edge of the network.

We have established that the nodes in the chain must have the dependency topology of N1, and, by nature of the chain, that each is to the left of N1 in the network. Since the network has only a finite number of stages, the chain must end at a node N' which is in the leftmost stage. N' must be blocked, by virtue of the type 1 dependency on it and its inclusion in the DDG. However, by assumption 1 above, N' has input on each of it's input channels. If blocked, it must have a type 2 dependency on its other output channel, and thus be a type L node.

A ghost message will thus be generated at N' which will propagate ghost messages or accesses on the chain of dependencies in question. Each node within the chain of dependencies is blocked waiting for input, and may thus propagate information on the output channels which form the chain when that input, in the form of a ghost message or access, arrives. This is a contradiction to our hypothesis, and thus lemma 4 is proved.

We now present our main theorem:

- Theorem 1: Ghost messages are sufficient to break any possible deadlock in a Local Synchrony implementation on a Banyan (equidistant) network.

By Lemma 1, there must be a non-empty set of nodes within the DDG having on their input channels one or more type 2 dependencies. Choose the node (N) from this set which has at the front of one of its input queues A1, the access of least timestamp. Node N cannot have type 2 dependencies on either of its output channels, because the nodes which those channels are connected to would thus have earlier timestamped accesses, by the conservative properties of Local Synchrony. If Node N has only type 1 dependencies or no dependencies on its output channels, then it must have a type 1 dependency on at least one input channel, or it would not be blocked. Thus, by the choice of node N, A1 must be at the head of the input queue of a channel which has a type 2 dependency on it. Node N is illustrated in figure 9.

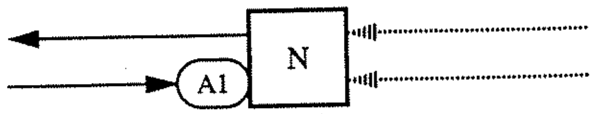

FIGURE 9. Node N, as chosen.

By Lemma 3 and the choice of node N, A1 has an earlier timestamp than *every* ghost message which will be generated by nodes in the DDG. By Lemma 4, a ghost message or an access is guaranteed to be propagated along the other input channel to node N. In either situation, progress is guaranteed, and node N will be able to make a routing decision. Each possibility is illustrated in figure 10.

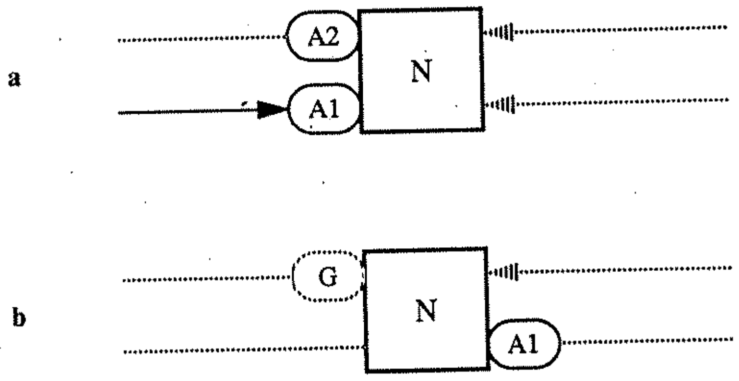

FIGURE 10. Progress at node N, due to ghost message propagation. a) Arriving access A2 denotes progress has already occurred. b) Arriving ghost message allows routing of access A1.

Note that, within the DDG, only ghost messages are generated after deadlock has arisen. As this is the case, the transfer of an access from node to node within the DDG denotes progress. Thus, if the information propagated to node N is an access (A2), the deadlock is already broken, and progress will continue as node N routes either A1 or A2. See figure 10a.

If the propagated information is a ghost message (G), A1 will be chosen by the router, as A1 has a lower timestamp, by lemma 3. Once chosen for routing, A1 will be able to be sent, since there are no type 2 dependencies on the output channels of node N. Thus progress is made by a node (N) within the DDG, and the deadlock is broken. See figure 10b.

5.0 An Analytical Model

In [Jen83], Jenq presents an analytical model for the performance of Banyan networks composed of 2-input, 2-output switching elements in a synchronous, single-buffered, packet-switching system. Jenq's model is based on the iterative steady-state solution of probabilistic equations governing the operation of the network, and provides measures for network throughput and packet delay. In this section, we present a similar model for the Local Synchrony implementation presented in section 3.

The network is assumed to be synchronous, where a clock cycle, t, consists of two phases, t_1 and t_2. During t_1, information relating to the state of input buffers passes from the last stage (at network outputs) backward to the first stage, so individual switches may make decisions as to which packets may be forwarded. Packet switching occurs during t_2. Jenq also makes the following simplifying assumptions:

1. Packets arriving at each input link are destined uniformly (or randomly) for output links at stage n.
2. The load on each input at stage 1 is uniform. Thus, at steady state, the load is balanced throughout the system.
3. If both buffers at a given switching element are full, and both packets are bound for the same output, then one is chosen at random to proceed.

In order to implement Local Synchrony in a single-buffered system, packets must arrive at the network inputs in sorted streams and be kept sorted as they traverse the network. We argue that an ordering which meets this condition can be generated without changing the random input streams assumed by Jenq.

In order to make this argument, we define a Local Synchrony timestamp for this implementation to have four components as discussed in section 3, in this order of importance: *pulse, combining, pid, tock*. The *combining* component of the timestamp will be the destination address of the packet. We shall assume that each output of the network represents a certain continuous range of destination addresses. Input packets are then uniformly distributed over both the outputs of the network and the range of destination addresses covered by each.

The *tock* component of the timestamp can be ignored, as it is impossible for packets from the same PE to travel separate paths or change their emitted order within the Banyan network. The *pid* component of the timestamp can be ignored also if we assume that exact packet collisions (where packets meet which are bound for the same destination, in the same pulse) do not occur. This simplifying assumption precludes the possibility of shared memory hot spots. Future work will take packet collisions into account.

Finally, the *pulse* component of the timestamp is applied randomly at each input. Packets in each input stream are given the same pulse component as long as their destination addresses are increasing, and the *pulse* component is incremented when the string of increasing destination addresses is broken.

A switching element in the network then makes a decision between two packets by passing that with the lower timestamp. Our assumptions have limited this decision to require only the *pulse* and *combining* components of the timestamp, and we have shown that both of these components are assigned based on the destination addresses of the packets, which are assumed to be random.

The choice made by a switching element concerning which packet to send next is based on a uniform random distribution of destination addresses. A given packet could have any destination, and a timestamp equal to or greater than the last input on that line.

The major difference between normal operation, modeled by Jenq, and Locally Synchronous operation, as we are modeling here, is that a switch with a packet on only one input channel may not pass that packet in the Locally Synchronous model. In such a situation, the switch must wait until both its input channels are full in order to make a conservative decision about which access to pass next.

Jenq introduces the following notation:

n = The number of stages in the switching network.

$p_0(k, t)$ = Probability that a buffer of a switching element at stage k is empty at the beginning of the tth clock period.

$p_1(k, t) = 1 - p_0(k, t)$

$q(k, t)$ = The load on an input buffer at stage k during the tth clock period. In other words, the probability that a packet is ready to be transmitted to the buffer during the tth clock period.

$r(k, t)$ = Probability that a packet in a buffer at stage k is able to move forward during the tth clock period, given that there is a packet in that buffer.

In a Local Synchrony implementation, the equations which describe these variables are as follows:

$$q(k, t) = .5 * p_1(k-1, t)^2 \qquad \text{for } k = 2, 3, 4, \ldots, n \qquad (1)$$

$$r(k, t) = .5 * p_1(k, t) * [p_1(k+1, t) * r(k+1, t) + p_0(k+1, t)] \qquad \text{for } k = 1, 2, 3, \ldots, n-1 \qquad (2)$$

$$r(n, t) = .5 * p_1(n, t)$$

$$p_0(k, t+1) = [1 - q(k, t)] * [p_0(k, t) + p_1(k, t) * r(k, t)] \quad (3)$$

$$p_1(k, t+1) = 1 - p_0(k, t+1) \quad (4)$$

This system, like Jenq's, converges to a steady state, with time-independent quantities $Q(k)$, $R(k)$, $P_0(k)$, and $P_1(k)$ for any k. Measuring the normalized throughput S and normalized mean packet delay d, with a uniform load of 1.0 on all inputs, we find that the system converges to values which indicate that performance of our model is slower than Jenq's by a factor of approximately four.

To interpret this result, we note that in general operation, a switch may route an access arriving on one of its input channels without regard to traffic on its other input channel. A switch operating under the conditions required for Local Synchrony, on the other hand, must conservatively route the access with the smallest timestamp. This means that a switch with one or more empty input channels *must* block. This new blocking requirement accounts for most of the negative effect on efficiency.

We note further that the model as presented does not take into account efficiency enhancments such as combining, which would effectively reduce traffic through the network. Although combining can be implemented in general operation, Local Synchrony *guarantees* that combinable accesses within the same logical time pulse will be combined, which may have a greater effect on traffic reduction. The model presented here also does not take into account the benefits of ghost messages, which, through faster dissemination of information, can increase system efficiency.

There is no doubt, however, that average throughput and delay times will be greater in a Local Synchrony implementation. However, Local Synchrony provides significant advantages in implementation. Isochrons allow concurrent atomic access to multiple shared memory elements without the need for locking protocols. Local Synchrony also guarantees sequential consistency for accesses, which generally allows PEs to pipeline accesses for better efficiency. It is our hypothesis that the power of Local Synchrony will offset its negative effects.

6.0 Conclusions

We have presented Local Synchrony, a synchronization model for shared memory parallel architectures using logical time. Local Synchrony is a conservative timestamp ordering protocol for shared memory accesses which implements a logical time system in an asynchronous architecture.

We have described an implementation of Local Synchrony on a Banyan equidistant network architecture consisting of 2-by-2 switching elements. Our description included discussion of the timestamp ordering system, the transport and routing mechanisms, and access flow control. Also, we pointed out that Local Synchrony facilitates the implementation of FIFO combining [Ran87], and discussed how this may be accomplished.

Because Local Synchrony is a conservative system, its operation differs from the norm for asynchronous architectures. The question of deadlock freedom becomes important, as blocking is possible in a switch. We have presented proof, using special information distributing messages called ghost messages, that Local Synchrony is deadlock free for this class of networks.

Finally, we presented an analytical model of our implementation which is similar to Jenq's [Jen83]. This model allows us to predict costs for Local Synchrony in comparison with an accepted analytical model.

BIBLIOGRAPHY

[BeG80] P. A. Bernstein and N. Goodman, "Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems", in *Proceedings of the 6th International Conference on Very Large Data Bases*, October 1980.

[BEG81] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", ACM *Computer Surveys 13* (1981), 185-222.

[ChM79] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs", *IEEE Transactions on Software Engineering SE-5*, 5 (September 1979), 440-452.

[GLR81] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph, "Coordinating Large Numbers of Processors", in *Proceedings of the 1981 International Conference on Parallel Processing*, 1981.

[Jef83] D. Jefferson, "Virtual Time", in *Proceedings of the 1983 International Conference on Parallel Processing*, 1983, 384-394.

[Jef85] D. Jefferson, "Virtual Time", *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 404-426.

[Jen83] Y. Jenq, "Performance Analysis of a Packet Switch Based on Single-Buffered Banyan Network", *IEEE Journal on Selected Areas in Communications SAC-1*, 6 (December 1983), 1014-1020.

[KRS88] C. P. Kruskal, L. Rudolph, and M. Snir, "Efficient Synchronization on Multiprocessors with Shared Memory", *ACM Transactions on Programming Languages and Systems 10*, 4 (October 1988), 579-601.

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

[Mil79] M. Milenkovic, *Update Synchronization in Multiaccess Systems*, University of Massachusetts, Amherst, May 1979.

[OwL82] S. Owicki and L. Lamport, "Proving Liveness Properties of Concurrent Programs", *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 455-495.

[Ran87] A. G. Ranade, "How to Emulate Shared Memory", in *Proceedings of the Annual Symposium on the Foundations of Computer Science '87*, IEEE, 1987, 185-194.

[RBJ88] A. G. Ranade, S. N. Bhatt and S. L. Johnsson, "The Fluent Abstract Machine", *Yale University Department of Computer Science Technical. Report 573*, January 1988.

[Ree83] D. P. Reed, "Implementing Atomic Actions on Decentralized Date", *ACM Transactions on Computer Systems 1*, 1 (February 1983), 2-23.

[ReF87] D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing*, MIT Press, Cambridge, Mass., 1987.

[RWW89] P. F. Reynolds, Jr., C. Williams and R. R. Wagner, Jr., "Parallel Operations", *UVA Computer Science Technical Report* 89-16, December 1989.

[ShS88] D. Shasha and M. Snir, Efficient and Correct Execution of Parallel Programs that Share Memory", *ACM Transactions on Programming Languages and Systems10*, 2 (April 1988), 282-312.

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

[Wil90] C. Williams, *Concurrency Control in Asynchronous Parallel Computations: A Research Proposal*, University of Virginia, 1990.