# Hardware Support for Parallel Discrete Event Simulations

Paul F. Reynolds, Jr.,
Carmen M. Pancerella

Computer Science Report No. TR-92-08
April 20, 1992

This research was supported in part by NSF Grant #CCR-9108448 and JPL Contract #957721.

# HARDWARE SUPPORT FOR PARALLEL DISCRETE EVENT SIMULATIONS

Paul F. Reynolds, Jr. and Carmen M. Pancerella

This document describes the functional requirements and hardware-level assumptions for special-purpose hardware to support efficient parallel discrete event simulation (PDES) [Fuji90] as well as other applications that can benefit from rapid dissemination of synchronization information. We begin by presenting a brief overview of the synchronization requirements of PDES and then describe in detail the requirements for hardware support. Following that, we describe a hardware framework that satisfies these requirements.

## 1. INTRODUCTION

Recently Reynolds [Reyn92] proposed a novel framework for providing rapid dissemination of critical synchronization information in all parallel discrete event simulation (PDES) [Fuji90] implementations. This framework can produce a significant reduction in the finishing times of parallel simulations. The strength of this framework lies in its use of special-purpose, high-speed hardware, specifically a *parallel reduction network* (*PRN*) which computes and disseminates results of *global reduction operations*. A global reduction operation is a binary, associative operation performed on data across all processors.

Current parallel architectures do not support the rapid computation and dissemination of results of binary, associative operations across all processors without a barrier synchronization. In these machines, all host processors must coordinate before a computation can begin. Our network separates the processors doing the application computation from those doing the synchronization computation; therefore, a barrier synchronization is not necessary to compute global operations. Furthermore, our hardware computes different binary, associative operations across state vectors of values in order to keep logically related data together, to capture consistent snapshots of the application, and to avoid race conditions.

In a system of a hundred processors, simple hardware based on the concept of a reduction tree could compute and disseminate results of binary, associative operations on the order of 100's of nanoseconds, 150 nanoseconds * $\log_2(n)$, $n = 100$ — three orders of magnitude faster than in typical distributed memory architectures — without interfering with the work of the application (host) processors. In terms of raw speed, the network continues to be beneficial for up to hundreds of thousands of processors. Since the network is performing reductions on the information fed into it, its advantage over typical host system communication times should extend into the range of millions of processors.

PRN's can disseminate global information many times faster than a communications network in a typical distributed memory multicomputer. We estimate that a global operation can be performed in 750 nanoseconds for a 32-processor machine using a PRN (assuming 150 ns. per PRN stage as described later). The Thinking Machines' CM-5 [Thin92] has comparable speeds for performing arithmetic reduction operations, yet a PRN can compute and disseminate global reduction results on state vectors *without* the coordination of the host processors. PRN's can be built and retrofitted to both existing parallel computers or to a tightly coupled network of high-speed processors for a fraction of the cost of building the computers themselves. PRN's will be completely independent of the host interconnection network of the parallel machine or the communications network in the distributed computing system. We propose using PRN's in conjunction with both shared memory and distributed memory parallel architectures as a means of rapidly disseminating a small amount of critical synchronization information in a PDES.

This paper is a companion paper to [Reyn92]; it describes requirements for a PDES and elaborates on the hardware implementation of the PRN that satisfies these requirements in accordance with established correctness criteria. In Section 2 we explain the requirements for PDES and briefly review the framework algorithms. In Section 3 we define the correctness criteria that are necessary in a hardware implementation. In Section 4 we present an abstract synchronization network. In Section 5 we outline the basic design of a pipelined PRN. In Section 6 we discuss the interface between a host processor and its corresponding auxiliary network processor.  In Section 7 we discuss the interface between the auxiliary processor and the PRN. We show how both interfaces are designed to satisfy all correctness criteria.  In Section 8 show how the PRN is also useful in enhancing a class of parallel algorithms that employ barrier synchronization. In sections 9 and 10 we discuss related synchronization networks.  Finally, we describe a prototype PRN which is currently being designed and built by the Departments of Computer Science and Electrical Engineering at the University of Virginia.

## 2.  REQUIREMENTS FOR PARALLEL SIMULATION

A general model of a PDES is introduced in [Misr86].  A PDES consists of a set of logical processes (LP's) that model physical processes in a system.  All interactions among physical processes are modeled by event messages sent among LP's.  Each message contains a timestamp indicating the logical time at which a scheduled event will occur.  Each LP maintains a relative counter, its logical clock, that denotes how far that LP has progressed.  It is very likely that each LP's logical clock will differ, and this asynchronous property of parallel simulation can lead to synchronization problems.

A *causality error* in a PDES occurs when event A depends on event B, and event A is executed before event B.  A parallel simulation will be correct if and only if each LP processes events in nondecreasing timestamp order. Adherence to this *local causality constraint* is sufficient, though not always necessary, to guarantee the absence of causality errors[Fuji90]. However, there is no way of guaranteeing that messages received by $LP_i$ occur in a specific order due to both the asynchronous nature of LP's logical clocks and communications delays in the host network.

There are two common approaches to PDES synchronization protocols:  *non-aggressive* and *aggressive*.  Non-aggressive protocols, as discussed in [ChMi79], must determine when it is safe to process an event in order to avoid causality errors. Aggressive approaches, the most well-known is Time Warp based on Jefferson's virtual time model [Jeff85], do not avoid causality errors but rather detect them and correct them using rollback.

The framework for PDES introduced in [Reyn92] is a low-level synchronization mechanism that sits below a PDES synchronization protocol. It disseminates a small set of globally reduced values. Operations are carried out in a synchronization network that is completely transparent to LP's.

The low-level framework for PDES computes four interesting global values across all LP's, $i = 1,...,n$, where $n$ is the number of LP's:

- $T'_\eta = MIN(T_{\eta_i})$: Minimum next event time
- $T'_\upsilon = MIN(T_{\upsilon_i})$: Minimum smallest unreceived message time
- $T'_\rho = MIN(T_{\rho_i})$: Minimum received but unacknowledged message time
- $T'_\tau = MIN(T_{\tau_i})$: Minimum acknowledged message time

These global values are computed from values maintained locally in each $LP_i$: 1) $T_{\eta_i}$: the logical time of the next event, 2) $T_{\upsilon_i}$: the smallest time-valued unreceived message, namely, the timestamp of the logically oldest outstanding unacknowledged message from $LP_i$, 3) $T'_\rho$: the smallest tagged message to acknowledge, and 4) $T'_\tau$: the smallest tagged acknowledgement to acknowledge. The globally reduced values for message acknowledgements were introduced in [Panc92].

Without loss of generality, it is assumed that each $LP_i$ occupies a unique physical processor $PP_i$. The values local to each $LP_i$, $T_{\eta_i}$, $T_{\upsilon_i}$, $T_{\rho_i}$, and $T_{\tau_i}$ are pipelined into a reduction network, and the network, in turn, computes the corresponding global values, $T'_\eta$, $T'_\upsilon$, $T'_\rho$, and $T'_\tau$. These global values, together with the following synchronization algorithms (which appear in [Reyn92] and [Panc92]), are sufficient to support deadlock-free parallel simulation, independent of the application being simulated.

```
TEST:     IF [($T_{\eta_i}$ = $T'_\eta$) AND ($T_{\eta_i}$ $\le$ $T'_\upsilon$) ]          --Identify the next event in system.
            THEN $LC_i$ := $T_{\eta_i}$;                              -- Set local clock.
                 Perform event; SENDMSG if needed;  --May send zero or more
                 Compute new $T_{\eta_i}$;                     -- messages.

SENDMSG:  [ IF  message time  < $T_{\upsilon_i}$                    --Update smallest unreceived message
              THEN $T_{\upsilon_i}$ := message time ]             -- value if necessary.
          Send Message, add [message time, message ID] to outstanding message sequence;

RCVMSG:   [ IF message time < $T_{\eta_i}$                        --Set next event time
             THEN $T_{\eta_i}$ := message time ];             -- if new msg is smallest valued.
          Add [message time, message ID] to received message sequence;
          [ IF $T_{\rho_i}$ = [$\infty$, –]                          --Set smallest unacknowledged msg
             THEN $T_{\rho_i}$ := [message time, message ID] ];      -- time if no other acks

CHKACK:   IF  ($T'_\tau$ = $T_{\rho_i}$) AND ($T_{\rho_i}$ $\ne$ [$\infty$,–])            --Receiver can remove ACK
             THEN Update $T_{\rho_i}$;
                  Remove message ID from received message sequence;
          IF ( $T'_\rho$ has been sent to me)               --Sender sees ACK and updates $T_{\upsilon_i}$
             THEN IF ( $T'_\rho$ is in outstanding message sequence)
                     THEN Mark message as acked in outstanding message sequence;
                          Remove any other acked ID's from outstanding message sequence;
                          $T_{\tau_i}$ :=  $T'_\rho$;
                          Update $T_{\upsilon_i}$, if necessary;
                     ELSE $T_{\tau_i}$ := [$\infty$,–];               --Sender can remove double ACK
             ELSE $T_{\tau_i}$ := [$\infty$,–];                       --Sender can remove double ACK
```

**Figure 1. Synchronization Algorithms.**

An instance of the algorithm seen in Figure 1 is executed asynchronously for each $LP_i$. The processes TEST, RCVMSG, and CHKACK execute concurrently, and SENDMSG is a subprogram of TEST. Furthermore, it is assumed that the order of changes to $T$ values in a given LP is preserved in global counterparts. In other words, if $LP_i$ changes $T_{\upsilon_i}$ and next changes $T_{\eta_i}$, the impact that $T_{\upsilon_i}$ has on $T'_\upsilon$ is seen by other LP's *no later than* the impact of $T_{\eta_i}$ on $T'_\eta$. This ordering constraint prevents race conditions which can cause the global $T$ values to reflect an incorrect global state. Finally, $LC_i$ is $LP_i$'s local clock.

In $LP_i$, TEST is assumed to execute at least each time either $T'_\eta$ or $T'_\upsilon$ changes value and the associated LP is blocked. In order to prevent deadlock, $LP_i$ must process an event if its local next event time is both the smallest in the system (i.e., $T_{\eta_i} = T'_\eta$) *and* no greater than the timestamp of the smallest outstanding message. This algorithm identifies the smallest next event time even when there are outstanding messages in the PDES. This feature allows deadlock prevention in a system when message traffic is always present.

When an LP sends an event message, the SENDMSG process maintains its sequence of [message time, message ID] pairs of unacknowledged messages. $T_{\upsilon_i}$ is maintained as the outstanding message with the smallest message time. When an LP receives an event message, the RCVMSG process adjusts $T_{\eta_i}$, if necessary, and an entry of [message time, global message ID] is made to the received message sequence, indicating a new message is received but unacknowledged. The local value of $T_{\rho_i}$ is updated immediately by $LP_i$ only if its current value is equal to [∞, —], indicating no message is unacknowledged. $T_{\rho_i}$ is an ordered pair of [message time, global message ID] such that a minimum operation is performed on message times across all LP's and the global message ID is a unique tag which identifies a particular message in the simulation.

In CHKACK *each* LP monitors the value of the global minimum unacknowledged message $T'_\rho$. The ordered pairs emerging from the reduction network are equivalent to acknowledgement messages. Once the ordered pair $T'_\rho$ is recognized by $LP_s$, the LP that sent the message and that message is in its outstanding message sequence, then $LP_s$ marks the corresponding message in its outstanding message sequence as acknowledged (but does not yet remove it from the sequence), removes all other messages that have been marked as acknowledged from its outstanding message sequence, modifies the value of $T_{\upsilon_s}$ accordingly, and sets $T_{\tau_s}$, the acknowledgement handshake. If $LP_s$ does not recognize the $T'_\rho$ as a message sent by it, $T_{\tau_s}$ is set to [∞,—], indicating no handshake. If $T'_\tau = T_{\rho_r}$, indicating that $LP_r$'s acknowledgement has been computed to be the minimum and also seen by the sender of the message, then $T_{\rho_r}$ is updated and the corresponding entry in the received message sequence pool is removed.

A synchronization network that supports this framework must allow individual processors of existing computers to asynchronously change local values that are reduced to compute corresponding global values which are then disseminated to all processors. The actual network is hidden from the applications software. The network should be able to be retrofitted to existing parallel computers or tightly coupled computing networks with little effort. The network must be easily realizable in hardware for a fraction of the cost of the machine itself. The network must rapidly perform minimum calculations on tagged local values from all processors, such that the corresponding tag of the minimum global value is equal to the tag of the minimum local value. The network must also compute results of other binary, associative operations. It must be completely independent of the host communications network. Finally, the network must operate under the correctness criteria listed in the next section.

## 3. CORRECTNESS CRITERIA

Inspection of the algorithms just given reveals that there are certain properties that must be ensured with respect to the order in which $T$ values local to a given LP are used in reduction operations in the reduction network. For example, in TEST and SENDMSG it is critical, when an LP completes an event, that a change to $T_{\upsilon_i}$ "be seen" no later than changes to $T_{\eta_i}$. By "be seen" we mean that the impact, if any, that a new $T_{\upsilon_i}$ has on its global counterpart, $T'_\upsilon$, should occur no later than when the new $T_{\eta_i}$ has an impact on $T'_\eta$. If this were not guaranteed then it would be possible for another LP, seeing changes to $T'_\upsilon$ and $T'_\eta$ out of order, to determine it has the smallest next event when in fact it does not.

There are two situations in which ordering must be preserved: when values enter the reduction network and when their globally reduced counterparts leave it. This suggests that the interface between the LP’s and the network must be designed with great care.

We note initially that there is no simple way to guarantee that an LP will see global $T$ values in exactly the same order in which they had local changes submitted to them on an input side of the network. For example, even if the network is designed to input local $T$ values from an $LP_i$ in the order in which $LP_i$ changes them, and therefore emit any changes to global $T$ values in the same order, when the network writes these values into output registers of a given $LP_i$, there is no simple way to guarantee that the values in the output registers will be read in the same order by $LP_j$. This is compounded further by the fact that a "well-intentioned" LP may not see changes in global values in the desired order. This can occur with the following sequence of events:

$LP_i$ changes $T_{\upsilon_i}$.

Reduction network reads $T_{\upsilon_i}$.

$LP_i$ changes $T_{\eta_i}$.

Reduction network reads $T_{\eta_i}$.

$LP_j$ reads old $T'_\upsilon$ (i.e., without impact of $T_{\upsilon_i}$).

Reduction network emits new $T'_\upsilon$ (i.e., with impact of $T_{\upsilon_i}$)

Reduction network emits new $T'_\eta$.

$LP_j$ reads new $T'_\eta$.

Note, in this example, that $LP_j$ is reading global $T$ values in the "proper order". Also note that $LP_j$ will not always know the "proper reading order", i.e., it may not be the case that $T_{\upsilon_i}$ is always changed before $T_{\eta_i}$. We conclude that we need to treat $T$ values as a *state vector* where we must guarantee that vectors of $T$ values are always valid states.

The atomicity and the ordering requirements of the framework algorithms shown in Figure 1 guarantee that an LP's local *T* values, i.e., its local state vector, *always* reflect a valid state; therefore, if the reduction network treats the global *T* values as a global state vector, then this global vector will always be a valid global state for a PDES.

We now define two possible ways in which an LP can modify its local *T* values. It is desirable that the hardware support these cases:

1) *atomic write without overwrite* — one or more of the local *T* values in the state vector are changed effectively simultaneously *and* further changes to the local state vector are not made until the network reads the local state vector; and

2) *atomic write with overwrite* — one or more of the local *T* values are changed effectively simultaneously; further changes to the local state vector may be made prior to the network reading the local state vector.

The framework algorithms are correct even if local *T* values are overwritten before they are used in the computation of global *T* values. Therefore, the framework algorithms can operate without overwrite (case 1), yet we recognize the second option as a desirable and relatively inexpensive capability which may be necessary for future synchronization algorithms. The framework does, however, require that a local *T* value be used in the computation of its global counterpart until a new value is assigned to that local value.

In the following sections, we develop a parallel reduction network (PRN) that is consistent with the requirements for PDES and also satisfies the correctness criteria that have just been established.

## 4. A SYNCHRONIZATION NETWORK

The block diagram of a network supporting efficient synchronization, as shown in Figure 2, was given in [Reyn92]. In this figure, we see each physical processor $PP_i$ contributing a set of input values to a synchronization network which in turn produces the results of four global reduction operations. (Since we are focusing on the hardware implementation, we discuss physical processors instead of logical processes in a PDES.) A global reduction operation is a binary, associative operation applied over all $n$ input values; in Figure 2, $n = 8$. Examples of global reduction operations include minimum/maximum calculations, logical operations (e.g., AND and OR), and summations.

In this example, we show four register pairs such that each pair consists of an input register and an output register. The input register contains a PP's operand for a global reduction operation, and the corresponding output register will eventually contain the result of that global

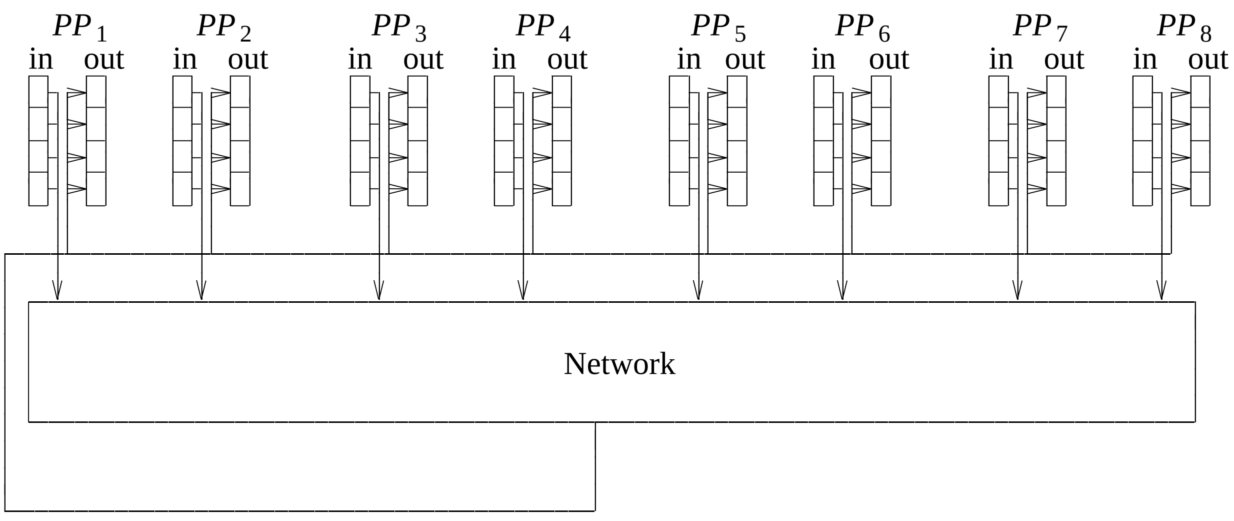

**Figure 2. An Abstract Synchronization Network.**

operation across all PP's. In general

$$out_{-,k} := \theta_k(in_{1,k}, in_{2,k}, \ldots, in_{n,k}) \quad \text{for} \quad k = 1, \ldots, m$$

where $m$ is the number of input value sets, $n$ is the number of PP's, $out_{-,k}$ represents the $k^{th}$ output variable in each of the PP's, and $\theta_k$ is the global reduction operator applied to the $k^{th}$ value set.

We now extend this abstraction to introduce a pipelined PRN. In Sections 6 and 7 we show how the pipelined PRN meets the correctness criteria discussed in Section 3. Furthermore, the proposed PRN does not limit the size of $m$ and is scalable for large $n$ since it is an $O(\log n)$ network.

## 5. A PIPELINED PARALLEL REDUCTION NETWORK

In Figure 3 we show the block diagram of a binary tree structured PRN supporting efficient synchronization. Each stage of the PRN consists of half as many ALU's as the stage above it, with the first stage consisting of half the number of processors. The PRN shown supports four inputs from eight PP's and produces four output global reductions. If the *in* and *out* registers are memory-mapped in the respective processors, the application process, in this case an LP in a parallel simulation, could store and retrieve the $T$ values without knowledge of the underlying PRN.

The interface to the PRN from each processor is identical. Each processor has sets of memory-mapped input registers and memory-mapped output registers. A processor can write to the input registers and read from the output registers; the PRN will read values from the input registers and write the corresponding global operation results into the output registers. This memory-mapped interface is a possible source of memory contention if both the PRN and the application process attempt to access the input or output registers simultaneously. Careful timing of accesses by the processor and the network can prevent this interference. In Section 7 we discuss how the interface between the processor and the PRN can be constructed in order to eliminate most memory contention.

Each node of the PRN is a simple arithmetic logic unit (ALU) which is capable of being programmed to compute a sequence of simple binary, associative operations. Each input and output register is actually a register pair consisting of a value register and tag register. These register pairs are fed into the PRN. If the operation is a *selective* operation, e.g. minimum or maximum, the tag of the "winner", e.g. the minimum or the maximum of two values, is sent with the result to the next level of the PRN. If the operation is non-selective, e.g. addition or logical

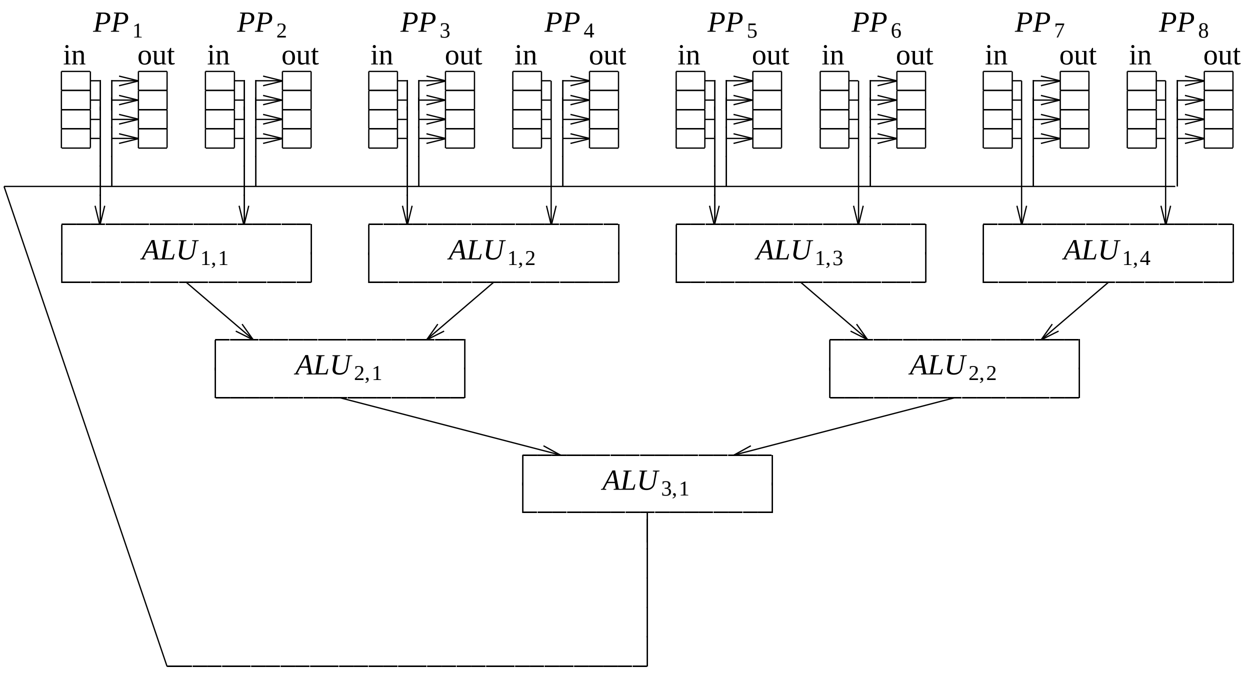

**Figure 3. A Pipelined Parallel Reduction Network.**

OR, the tag which is sent to the next level is chosen deterministically. Because of the PRN's binary tree properties, a global reduction operation can be computed and disseminated in $O(\log n)$ time using the PRN.

A *PRN major cycle* is the time required for a value to be input into the PRN and its corresponding output value to exit the PRN. A *PRN minor cycle* is the time required for a value to move from one stage of the pipelined PRN to the next. A PRN major cycle is therefore equal to log *n* minor cycles. A *complete PRN input cycle* is the time required for *m* values to be input to the PRN; a *complete PRN output cycle* is the time required for *m* output values to exit the PRN, where *m* is the total number of input registers to the PRN. (A complete input cycle and a complete output cycle should be approximately equal; the functionality, however, is different: the input cycle assumes *m* values are read and the output cycle assumes *m* values are written.) We estimate that a PRN minor cycle, the time required for computation of one reduction operation plus the delay incurred between stages, will take 150 ns., assuming current hardware technology.

Pipelining is employed in order to use this network efficiently: the partial results will be pipelined through the log *n* stages of the PRN such that each ALU is always busy. By pipelining sets of values through the PRN, a set of updates for all output registers is available, assuming steady state of the PRN, every *mc* units, where *m* is the number of input value sets and *c* is the minor cycle time. (There is an added transitional cost of *C* for network startup, where *C* is the major cycle time.) The PRN can be viewed as a synchronous network that continually produces results of *m* global reduction operations, where each physical processor has *m* input registers to the PRN. The tree structure of the PRN facilitates the pipelining since a tree circuit is easier to synchronize than other structures [Blel89]. The PRN, however, operates asynchronously to the PP's, and it does not block if an input register has not been updated during the previous major cycle.

Each node of the PRN operates asynchronously. As input values and their corresponding operation code are available, the ALU computes a resulting value and outputs that value. This asynchronous design facilitates the addition of floating point processors at each ALU node. In this case a floating point operation forms a "bubble" in the pipeline. Another advantage of the asynchronous node design is the PRN is scalable to large sizes since communication between nodes in the PRN can controlled with hardware handshakes. This eliminates the need for a central clock in the PRN, so there is no concern about clock skew in a large network.

A PRN can be constructed with any number of input registers; however, if there are less than $\log n$ registers, where $n$ is the number of processors, it is guaranteed that a global reduction on each input register will be performed within one major cycle of the PRN. If there are more than $\log n$ input registers, then all global synchronization values are not calculated and disseminated within one major cycle of the PRN. For example, if there are $O(n)$ input registers and $n$ is sufficiently large, then the additional delay in disseminating information can approach the expected communications delay in a typically slower host network and the PRN becomes a bottleneck rather than a performance enhancement.

We expect a PRN to be beneficial to a shared memory multiprocessor, a distributed memory multicomputer, and a tightly coupled network of high-speed processors as a complementary network to the host communications network. In a distributed memory machine, such as a hypercube, the time required to disseminate global information can be orders of magnitude more than the cost of performing a simple floating point operation, due only to delays in the communication network. The PRN can reduce the amount of message traffic needed in a computation; furthermore, global synchronization information can be distributed to all processors much faster. In a shared memory machine, the frequent update of a small amount of global information can be a source of memory contention or hot spots [PfNo85]. The PRN eliminates the synchronization traffic both in the interconnection network and at the global memory.

## 6. AN AUXILIARY NETWORK PROCESSOR

Auxiliary, general-purpose processors are added at the interface of the reduction network, one processor per host processor, in order to manage the high-speed I/O with the PRN. The high-speed synchronization activity is moved to these dedicated processors in order to increase the efficiency of synchronization algorithms such as the PDES framework. The *host processor* is solely responsible for application processing, e.g., executing events and sending/receiving event messages in a parallel simulation. The *auxiliary processor* (*AP*) is responsible for executing synchronization algorithms, i.e., the framework algorithms in Figure 1, and for interfacing with both the parallel reduction network (PRN) and the host processor. The PRN and the synchronization algorithms are completely transparent to the host processor.

A block diagram of the interface between a host processor and its auxiliary processor can be seen in Figure 4. The AP is programmable and will execute synchronization algorithms that have been loaded into its program store. This processor writes to the PRN input registers and reads the PRN output registers; the specific interface between the reduction network and the AP is discussed in the next section. The auxiliary processor monitors the reduction network and writes the global synchronization values into single cell registers readable by the host processor; the host processor executes the TEST process from the framework algorithms and performs all simulation events. The single cell registers are treated as a single state vector output from the reduction network. The host processor can compute *global virtual time* (*GVT*) [Jeff85] and make adaptive processing decisions based on the global synchronization values.

In addition to executing the SENDMSG and RCVMSG algorithms in Figure 1, the auxiliary processor executes all message acknowledgement algorithms, such as CHKACK in Figure 1. This increases the speed at which messages can be acknowledged through the PRN. Efficient message acknowledgements through the reduction network was a motivating force behind the

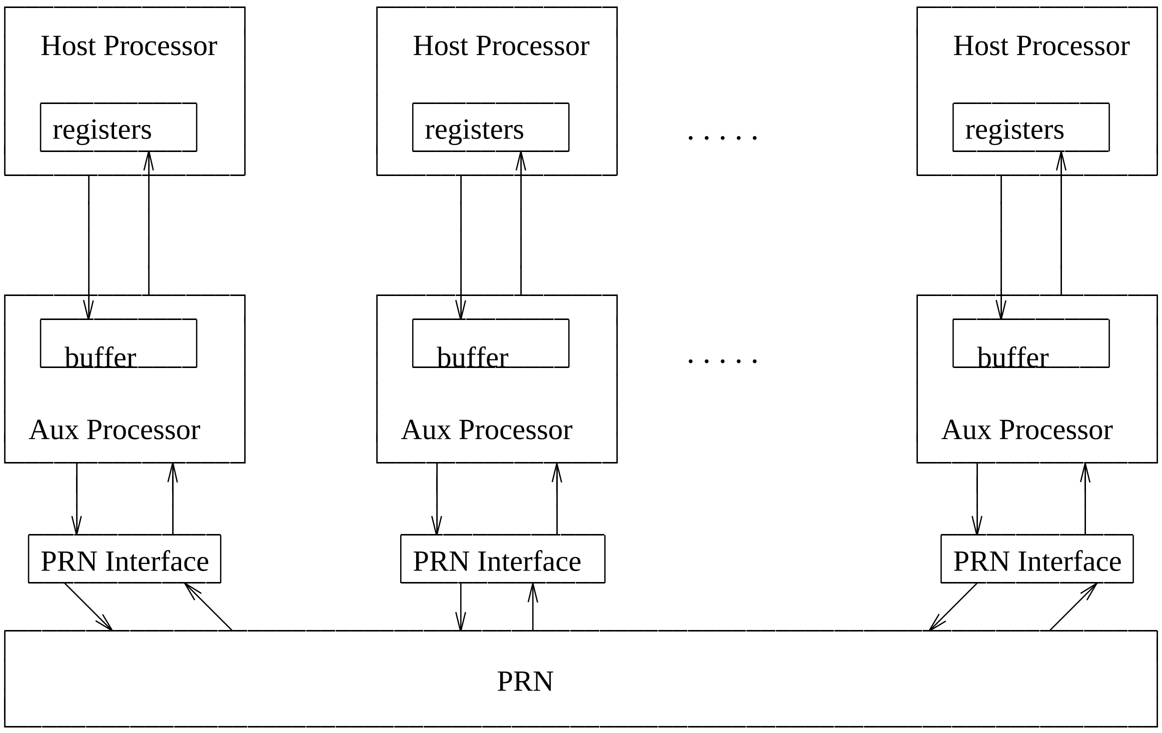

**Figure 4. A Host Processor — Auxiliary Processor Pair.**

addition of a dedicated processor at the output stage of the PRN. This is discussed in greater detail in [Panc92].

As seen in Figure 4, an ordered buffer of tagged messages is used for communication from the host processor to the auxiliary processor. The host processor writes into the buffer while the AP reads from the buffer. Entries into the buffer preserve ordering and are tagged so that the auxiliary processor correctly processes the information. We can identify three cases in a PDES operating within the framework where the host processor must communicate to the auxiliary processor: when an event message is sent, when an event message is received, and when the next event time changes as a result of the completion of an event or receipt of a message. We expect these to be low frequency events as compared to the frequency of I/O activity between the auxiliary processor and the PRN.

The addition of auxiliary processors at the input and output stages of the PRN mainly facilitates the management of high-frequency data coming out of the PRN. The auxiliary processors are responsible for inserting data into the PRN as well but with much lower frequency. By adding auxiliary processors the interface between the PRN and the host processors is as simple as possible. Furthermore, the portability of the reduction network to other parallel processors is enhanced.  We assume each AP boots up in a state in which it ultimately listens to its host processor interface.  The host processor sends its auxiliary processor data representing a program to load and execute.  The mechanism for downloading these programs can be kept simple.

We assume that one of the host processors in the system and its corresponding auxiliary processor is designated as a master pair of processors.  The master HP communicates PRN programming information to a state machine which controls the PRN.  Critical information to be passed to the state machine includes the number of components in a state vector and the operations to be performed on components; for example, it can be specified that all first components are to be summed, all second components OR'ed, and the minimum it to be taken on all third components in a three component state vector.

Note that the master host processor can send new PRN programming information to its auxiliary processor at any time.  Similarly, host processors can pass data to their respective auxiliary processors indicating that they are to receive new programs to execute.  This will permit dynamic reprogramming of the AP's and the PRN.  We assume that applications running on the HP's and programs running on the AP's are sufficiently robust to support this dynamic reprogramming.

## 7. INTERFACE BETWEEN AN AUXILIARY PROCESSOR AND THE PRN

As mentioned in Section 3, there are certain properties that must be ensured with respect to the order in which values local to a given AP are used in PRN reduction operations in order to correctly support the PDES framework.

The network interface between the PRN and each processor, depicted in Figure 5, is identical. It is designed to treat both local input values and global output values as state vectors. It preserves the validity of state vectors both when the local values enter the reduction network and when their globally reduced counterparts leave it. Values are viewed as vectors since the network reads the input registers one by one as a group (if the local input values represent a valid local state) and emits the output registers as a reduction on the same group (so that the global output values will represent a valid global state). Valid state vectors are guaranteed by clocking a valid local state vector into the registers read by the PRN after every complete PRN input cycle and clocking a valid global state vector into the registers read by an AP after every complete PRN output cycle. Without loss of generality, we assume a complete PRN cycle begins with register $R_{m-1}$ and ends with register $R_0$.

We now discuss the interface on the input side of the PRN; as seen in Figure 5, there are three rows of input registers. The first row of registers is written by $AP_i$, and any of these registers may be written zero or more times prior to being fed into the PRN, provided that the state vector is overwritable. The third row of registers is read by the PRN. The second row is an intermediate row of input registers which facilitates both performing non-overwrite writes and capturing snapshots of valid local state vectors. When $AP_i$ completes its write(s) to its input registers, it sets a valid bit, indicating that the AP input registers contain a valid state and can be clocked into the intermediate input registers. An auxiliary processor cannot write to its AP input registers until the valid bit is reset to zero, indicating that the intermediate input registers have

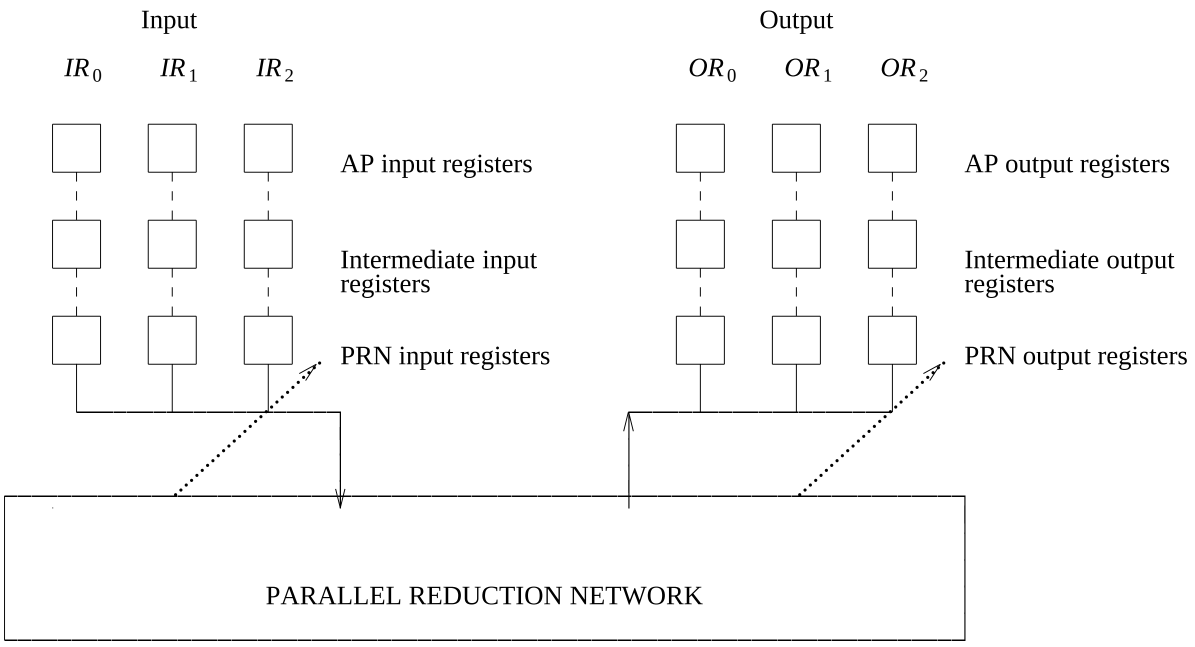

**Figure 5. Interface between an auxiliary processor and the PRN.**

been written. If the write is non-overwritable, the AP also sets an overwrite bit that follows the AP input registers into the intermediate input registers when the AP registers are clocked into the intermediate registers; the intermediate registers will not be updated again until they are clocked into the PRN input registers. An AP, therefore, does not need to halt its processing once the intermediate registers are set for non-overwrite until a second write to the the AP input registers occurs. Due to the expected speed of the register banks, we do not expect AP's to ever have to wait. The AP input registers are only clocked into the intermediate input registers when they represent a valid state (i.e., the valid bit is set). The coordination between the AP input registers and the intermediate input registers is accomplished by a hardware handshake mechanism. Intermediate input registers are only clocked into the PRN input registers at the end of a complete

PRN input cycle, depicted by the dotted arrow on the input side in Figure 5. The registers never block the PRN, and flow through the three register banks guarantees this. The PRN will read the PRN input registers nondestructively since we expect it will read them multiple times due to speed differences between the PRN and the slower auxiliary processor. Both the intermediate and the PRN input registers represent valid state vectors of $AP_i$ at all times because if the AP input registers contain a valid state vector when the valid bit is set after an atomic write, the registers preserve the integrity of the state vector.

In Figure 5 there are also three rows of registers on the output side on the PRN. The first row of registers is read by $AP_i$. The third row of registers is written by the PRN. The second row of registers is an intermediate output register bank that is used to prevent interference between $AP_i$ and the reduction network. The PRN output registers are only clocked into the intermediate registers at the end of a complete PRN output cycle, depicted by the dotted arrow on the output side in Figure 5. Intermediate output registers can be clocked into the AP output registers whenever the AP is done reading the state vector represented by the AP output registers. Intermediate output registers are not necessary for algorithm correctness. If intermediate registers were not employed, the PRN would have to skip writing a new vector into the AP output registers until the AP has read the entire state vector from the AP output registers. Both the intermediate and AP output registers represent valid global state vectors at all times; the PRN output registers contain a valid global state only at the end of a complete PRN output cycle. The output sweep is not synchronous to the input sweep; the sweep differs by log $n$ modulo $m$, where $m$ is the number of inputs to the PRN.

The PRN continuously writes values into the PRN output registers. Once a valid state vector is formed in the PRN output stage, a line is raised to enable the clocking mechanism between the PRN stage and the intermediate stage and at the same time, disable the clocking mechanism between the intermediate output stage and the AP output stage. However, the actual writing of intermediate registers takes place after some small delay, to allow any clocking between the intermediate and AP stages, which may have already been initiated, to complete. The line is then lowered, disabling the clocking mechanism between the PRN output registers and the intermediate output registers and enabling the clocking mechanism between the intermediate output registers and the AP output registers. This line serves as a master control to the individual clocking mechanisms of each set of AP registers; disabling it will disable all of them and enabling it will enable all of them. The clocking mechanism for each set of AP registers is enabled by this master line as well as a line controlled by the auxiliary processor. Thus, when an auxiliary processor wishes to read its processor output registers, it disables the clocking mechanism for its processor output registers by lowering the line. This may cause the processor to miss some of the outputs, but under most synchronization algorithms, this is acceptable.

## 8. APPLICATIONS OF THE PRN BEYOND PARALLEL SIMULATION

We have introduced the PRN as a hardware network supporting PDES. Besides supporting the parallel simulation framework, there are several ways in which a PRN can support PDES implementations. Non-aggressive protocols can be enhanced by disseminating minimum *lookahead* [Fuji88] values through the PRN. Iterative PDES algorithms often require the rapid computation and dissemination of ceiling values or fault values. (See [Dick90], [Luba88], [Luba89], [SoSH89], [SoBW88], [Ayan89].) Aggressive protocols [Jeff85] can be enhanced by the efficient computation of GVT and by limiting aggressive processing without the simulation deadlocking. Furthermore, by rapidly disseminating GVT and by limiting aggressive processing, the PRN can benefit a PDES that uses the rollback chip [FuTG88a, FuTG88b], a high-speed memory device which enhances a Time Warp simulation; the PRN provides accurate state information which facilitates fossil collection on the rollback chip. Finally, global termination conditions (e.g. sums and boolean operations) in a PDES [AbRi91] can be calculated and disseminated in the PRN.

The applicability of PRN's is not limited to parallel simulation; PRN's can be efficiently used in many application areas that require the rapid dissemination of global synchronization results. For example, the PRN can support a variety of parallel numerical computations. One is solving linear equations using numerical iterative algorithms, such as the Jacobi iterative method or the SOR iterative method. The PRN can be used to disseminate information about a process's iteration number or convergence information. A local iteration counter, $IC_i$, can be submitted to the PRN and a process can block or continue based on how many iterations ahead of the global minimum iteration counter $IC'$ it has advanced. Any iterative numerical method needs a convergence test; the PRN can efficiently perform and disseminate convergence tests without using message broadcasts. High-speed convergence evaluation can be easily accomplished as follows. Each PP is performing iterative calculations, a convergence evaluation of internal errors in each PP is obtained by computing the maximum error within each PP, this local error is submitted to the PRN, a global maximum error is computed using the PRN, and this global error is compared to the criterion for the convergence test. Other convergence tests require each PP to perform a local convergence test and a global convergence test is performed as the logical-AND of all tests; the PRN supports this logical computation.

Another numerical method that can be performed using the PRN is an inner product computation. On a parallel machine, an inner product computation typically requires a fan-in computation followed by a broadcast of the result. The inner product computation and broadcast can be done efficiently in the PRN. The conjugate gradient method, which is a popular iterative method for solving linear equations, is an example of an algorithm which benefits from an efficient inner product computation. The PRN can also support the same numerical applications for which the Finite Element Machine (FEM) [CrKn85] was developed.

The PRN as defined above can also support barrier synchronization as defined in [Ston90]. Each processor can submit a flag to the PRN, and all processes can block until the global reduction operation (a logical operation, in this case) reflects when the barrier has been reached by all processors. The PRN also supports barrier synchronization among a set of all processors since non-participating processors can simply contribute a positive flag to the logical reduction operation and not block execution. Furthermore, the PRN supports multiple barriers [Ston90] efficiently without additional hardware [HeRS89] since the PRN already has a set of input registers.

A PRN-like network can support an approach to load sharing we call *Waterfall Load Sharing* (WFLS). WFLS takes into account the distance a relocated process's remote data accesses (back to its home processor) must traverse. So, given a linear sequence of three processors with the following loads: (2 1 0), a conventional load sharing algorithm would take the surplus process from the first processor and give it to the third, creating the load (1 1 1). WFLS would achieve the same distribution but by taking the first processor's surplus and assigning it to the second processor and by taking the second processor's original process and assigning it to the third processor. The difference between the two approaches is that with the conventional approach one process must access data on its home processor across two communication links, while with WFLS there are two processes, each of which must access its data one communication link away. There are cases where WFLS will produce shorter finishing times for a set of processes than that produced by a conventional approach.

WFLS algorithms can use the PRN to provide near perfect state information about processor loads. Furthermore, the PRN can be used to support the computation of a global minimum load, a global load average, as well as more "target specific" information for subsets of processors. These values can be used to implement a WFLS algorithm. The design of the PRN as presented here does not support the generation and dissemination of target specific information in the general case. However, there are cases where it does and these cases may be good enough to support WFLS. Of courses, advances made in dissemination of target specific of information would be beneficial as well. As noted, this latter activity is an open research issue.

The trend in distributed memory parallel architectures is a mesh-connected communication topology; Intel's Touchstone DELTA machine [Lill91] is one such example. While this architecture is scalable to a larger number of processors than a hypercube, the number of hops between two processors can exceed $\log n$, where $n$ is the total number of processors. This suggests a greater need for the efficient dissemination of global synchronization information in the future.

## 9. HARDWARE SUPPORT FOR PARALLEL SIMULATION

The computation requirements for PDES continue to grow. The simulation of large communication networks and battlefield scenarios, for example, both require a significant amount of computation time. In addition, simulation programs, especially those employing aggressive processing, often utilize a large amount of memory. Therefore, future research in the area of hardware support for PDES is important.

The use of special-purpose hardware to improve the performance of simulation programs is not novel. Lubachevsky [Luba88] suggests using a binary tree implemented in hardware in order to support synchronization barriers and to compute and broadcast a minimum next event time in a bounded lag PDES. His control synchronization network is presented strictly in support of the bounded lag protocol; nonetheless, this has served as a motivating factor in our approach.

Hardware enhancements for Time Warp have been prevalent in the research; for example, Livny and Manber suggested using token rings for disseminating GVT [LiMa85]. The current trend in this area of research, however, is to design a high performance, discrete event simulation engine. Fujimoto initially targeted the Virtual Time Machine [Fuji89] as hardware support for discrete event simulation, but this machine is now intended to utilize an aggressive style of execution in a general purpose parallel computer. Fujimoto, *et. al.*, developed the *rollback chip* [FuTG88a, FuTG88b] as a hardware enhancement to a Time Warp engine. The rollback chip is a memory management unit that facilitates the state saving and restoration that is inherent in aggressive protocols such as Time Warp. As reported in [BuRo90] the chip has excellent performance capabilities.

At about the same time that Reynolds introduced the framework, Filoque, *et.al.*, [FiGP91] proposed the use of a processor network with programmable logic for efficient global computations, such as the computation of GVT in a Time Warp simulation. This hardware is not a single network like the PRN; it is, however, a distributed system of *sockets*, one per processor. The reprogrammable sockets are connected in a pipelined ring, forming the computation engine. A token is inserted into the ring by a designated control socket. It travels around the ring, performing partial computations at each socket. When the token returns to the controller, the global computation is complete. Therefore, their proposed hardware performs global computations in $O(n)$ time whereas the PRN performs the same computations in $O(\log n)$ time. Furthermore, the proposed synchronization algorithms for computing GVT in [FiGP91] rely on the host communication network for message acknowledgements and our framework uses the PRN. The goals of both approaches are similar, but our framework is more efficient, more flexible, and more scalable.

## 10. RELATED SYNCHRONIZATION NETWORKS

The proposal of using a separate synchronization network for improving system performance is not new. The IBM RP3 [PfBG85] was designed as a shared memory multiprocessor that houses both a combining network for synchronization traffic and a low latency network for regular message traffic. Our proposed special-purpose hardware is not as complex or expensive as a combining network, yet it performs global synchronization operations very efficiently.

The Finite Element Machine, a NASA prototype, utilizes a binary tree-structured max/summation network to perform the global sum and maximum calculations necessary to support structural analysis algorithms based on the finite element method [CrKn85, JoSc79]. Like the hardware we propose, the sum and max calculations in the FEM are calculated alternately without processor synchronization. Our hardware design, however, employs a small set of input and output registers, whereas the network in the FEM uses a single input and a single output register. Furthermore, our proposed synchronization network is also enhanced to satisfy correctness criteria for our PDES framework.

Global operations on the Intel iPSC/2 [Inte89] are provided for arithmetic and logical binary associative operations where each processor contributes a value and all processors receive the result. They require the complete synchronization of all processors; all processors must contribute a new value to each global operation and a global operation blocks until all processors enter one. Our approach is asynchronous (i.e., allows stale data to be contributed to global operations).

The CM-5 [Thin92] contains two separate networks for different types of communication and synchronization: the data network is the primary message-passing network in the machine and the control network provides hardware support for common *cooperative operations*. The control network supports "soft" barrier synchronization, arithmetic and logical reduction operations, parallel prefix operations, and segmented parallel prefix operations. Like the Intel iPSC/2, the reduction operations require processors to call global operation functions with specific values; in contrast our PRN performs global operations continuously. Furthermore, our hardware design employs auxiliary processors to manage the high-speed data emitted from the PRN, and the PRN computes reductions on state vectors.

Several researchers have proposed the use of hardware to implement barrier synchronization. In Hoshino's PAX Computer [Hosh85] barrier synchronization is accomplished by each processor setting a single bit such that the collection of bits is fed into an AND gate and an OR gate, and the outputs of both gates are bussed to all processors. Stone [Ston90] suggests the use of global busses to find the maximum value in a set and to implement Fetch-and-increment as a synchronization technique. Researchers at IBM have constructed a configuration of barrier synchronization modules [HeRS89] as a low-cost device for barrier synchronization. The hardware that we propose, on the other hand, provides support for a larger class of algorithms than barrier synchronization algorithms.

Blelloch [Blel89] proposed a tree-structured hardware implementation of parallel prefix operations. One of our future goals is to enhance our hardware design to calculate and disseminate "target specific" synchronization information in a PDES. Parallel prefix computations may be useful in realizing that goal.

## 11. A PROTOTYPE PRN

We are in the process of building a prototype PRN, of size 4, which interfaces with a Sparc Cluster. The Sparc Cluster contains Sparc processors connected to a VME backplane. The interface between the Sparc processor and the auxiliary processor will have an S-bus connection and a dual-ported RAM interface. Management of the dual-ported RAM is through software resident on host and auxiliary processors. The S-bus has a bandwidth of about 100 megabytes for 32-bit words. We expect a throughput of 25 MB/second from host to auxiliary processor.

The auxiliary processor is a 32-bit general purpose processor, specifically a 25 MHz Motorola 68020. Each auxiliary processor will have one megabyte of RAM. The interface between an auxiliary processor and the PRN contains input and output state vectors of size 8, where each element of the vector is a 32-bit value/32-bit tag pair.

The PRN provides 32-bit fixed point arithmetic. The ALU's require 40 nanoseconds for a fixed point addition. The PRN requires two operations to perform selective operations such as minimum and maximum. In a selective operation, the tag of the "winner", e.g. the minimum or the maximum of two input values, is passed to the next level of ALU's. One minor cycle of the PRN will take 150 nanoseconds.

## 12. CONCLUSIONS AND FUTURE WORK

We have introduced an augmented parallel reduction network for disseminating small amounts of global synchronization information in a parallel MIMD computer. This network is easily realized in hardware, cost-efficient, and scalable. Furthermore, this network can be useful in a shared memory multiprocessor or a distributed memory multicomputer. In [Reyn92], it was shown how the PRN can effectively support PDES. We have also shown how the PRN can support parallel numerical calculations and barrier synchronization. We feel that this network can effectively improve the running time of many other loosely synchronous computations [FoJL88] that periodically require a small amount of global synchronization information to be disseminated.

The novel features of this network are a network interface which preserves state vectors and the use of an auxiliary processor to manage high-frequency data asynchronously to the host processor's computation. We predict that the framework algorithms and the PRN will produce a significant reduction in the finishing times of parallel simulations.

A PRN, as described in this paper, provides hardware support for *global* synchronization. Many PDES's can benefit from *target specific* synchronization since simulation topologies are often static and LP's typically communicate with a small set of LP's. In the future, we plan to investigate hardware support for the rapid dissemination of target specific synchronization information.

## 13. REFERENCES

[AbRi91] Abrams, M. and Richardson, D., "Implementing a Global Termination Condition and Collecting Output Measures in Parallel Simulation", *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, Anaheim, California, pp. 86-91, (January 1991).

[Ayan89] Ayani, R., "A Parallel Simulation Scheme Based on Distances Between Objects", *Proceedings of the SCS Multiconference on Distributed Simulation*, Tampa, Florida, pp. 113-118, (March 1989).

[Blel89] Blelloch, G. E., "Scans as Primitive Parallel Operations", *IEEE Transactions on Computers*, Vol. 38, No. 11, pp. 1526-1538, (November 1989).

[BuRo90] Buzzell, C. A. and Robb, M. J., "Modular VME Rollback Hardware for Time Warp", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 153-156, (January 1990).

[ChMi79] Chandy, K. M. and Misra, J., "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs", *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 5, pp. 440-452, (September 1979).

[CrKn85] Crockett, T. W. and Knott, J. D., "System Software for the Finite Element Machine", NASA Contractor Report 3870, NASA Langley, Hampton, Virginia, February 1985.

[Dick90] Dickens, P. M., "An Analytic Investigation of the Global Rollback Algorithm", A Research Proposal, Department of Computer Science, University of Virginia, Charlottesville, Virginia, September 1990.

[FiGP91] Filoque, J. M., Gautrin, E. and Pottier, B., "Efficient Global Computations on a Processors Network with Programmable Logic", Report 1374, Institut National de Recherche en Informatique et en Anutomatique, France, January 1991.

[FoJL88] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J. and Walker, D., **Solving Problems on Concurrent Processors, Volume 1**, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1988.

[Fuji88] Fujimoto, R. M., "Lookahead in Parallel Discrete Event Simulation", *Proceedings of the 1988 International Conference on Parallel Processing*, University Park, Pennsylvania, pp. 34-41, (August 1988).

[FuTG88a] Fujimoto, R. M., Tsai, J. and Gopalakrishnan, G. C., "Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp", Technical Report No. UUCS-88-011, Department of Computer Science, University of Utah, Salt Lake City, Utah, July 1988.

[FuTG88b] Fujimoto, R. M., Tsai, J. J. and Gopalakrishnan, G., "The Roll Back Chip: Hardware Support for Distributed Simulation Using Time Warp", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 81-86, (February 1988).

[Fuji89] Fujimoto, R. M., "The Virtual Time Machine", *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, Santa Fe, New Mexico, pp. 199-208, (June 1989).

[Fuji90] Fujimoto, R. M., "Parallel Discrete Event Simulation", *Communications of the ACM*, Vol. 33, No. 10, pp. 30-53, (October 1990).

[HeRS89] Heidelberger, P., Rathi, B. D. and Stone, H. S., "A Low-Cost Device for Contention-Free Barrier Synchronization", *IBM Technical Disclosure Bulletin*, Vol. 31, No. 11, pp. 382-389, (April 1989), IBM Research.

[Hosh85] Hoshino, T., **PAX Computer: High-Speed Parallel Processing and Scientific Computing**, Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

[Inte89] Intel Corporation, **iPSC/2 Programmer's Reference Manual**, Intel Scientific Computers, Beaverton, Oregon, October 1989.

[Jeff85] Jefferson, D. R., "Virtual Time", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404-425, (July 1985).

[JoSc79] Jordan, H. F., Scalabrin, M. and Calvert, W., "A Comparison of Three Types of Multiprocessor Algorithms", *Proceedings of the 1979 International Conference on Parallel Processing*, pp. 231-238, (August 1979).

[Lill91] Lillevik, S. L., "The Touchstone 30 Gigaflop DELTA Prototype", *Proceedings of the Sixth Distributed Memory Computing Conference*, Portland, Oregon, pp. 671-677, (1991).

[LiMa85] Livny, M. and Manber, U., "Distributed Computation Via Active Messages", *IEEE Transactions on Computers*, Vol. C-34, No. 12, pp. 1185-1190, (December 1985).

[Luba88] Lubachevsky, B. D., "Bounded Lag Distributed Discrete Event Simulation", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 183-191, (February 1988).

[Luba89] Lubachevsky, B. D., "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks", *Communications of the ACM*, Vol. 32, No. 1, pp. 111-123, (January 1989).

[Misr86] Misra, J., "Distributed Discrete-Event Simulation", *Computing Surveys*, Vol. 18, No. 1, pp. 39-65, (March 1986).

[Panc92] Pancerella, C. M., "Improving the Efficiency of a Framework for Parallel Simulations", *to appear in the Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, Newport Beach, California, (January 1992).

[PfNo85] Pfister, G. F. and Norton, V. A., "’Hot Spot’ Contention and Combining in Multistage Interconnection Networks", *IEEE Transactions on Computers*, Vol. C-34, No. 10, pp. 943-948, (October 1985).

[PfBG85] Pfister, G. F., Brantley, W. C., George, D. A., Harvey, S. L., Kleinfelder, W. J., McAuliffe, K. P., Melton, E. A., Norton, V. A. and Weiss, J., "The IBM Research Parallel Prototype (RP3): Introduction and Architecture", *Proceedings of the 1985 International Conference on Parallel Processing*, St. Charles, Illinois, pp. 764-771, (August 1985).

[Reyn92] Reynolds Jr., P. F., "An Efficient Framework for Parallel Simulations", *to appear in International Journal on Computer Simulation*, (1992).

[SoBW88] Sokol, L. M., Briscoe, D. P. and Wieland, A. P., "MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 34-42, (February 1988).

[SoSH89] Sokol, L. M., Stucky, B. K. and Hwang, V. S., "MTW: A Control Mechanism for Parallel Discrete Simulation", *Proceedings of the 1989 International Conference on Parallel Processing*, University Park, Pennsylvania, pp. 34-41, (August 1989).

[Ston90] Stone, H. S., **High-Performance Computer Architecture**, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[Thin92] Thinking Machines Corporation, **The Connection Machine CM-5 Technical Summary** , Thinking Machines Corporation, Cambridge, Massachusetts, January 1992.