*Proceedings of the 1992 Winter Simulation Conference*
*ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson*

# MAKING PARALLEL SIMULATIONS GO FAST

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

Department of Computer Science
School of Engineering and Applied Science
Thornton Hall
University of Virginia
Charlottesville, Virginia 22903, U.S.A.

## ABSTRACT

Parallel simulations executing on a closely coupled network of processors require synchronization of logical processes across all processors. This synchronization overhead requires coordination of host processors and additional message traffic in the host network. We introduce specialized hardware to offload all parallel simulation synchronization overhead from host processors and the host network in a closely coupled network. In this hardware design, critical synchronization information is disseminated in the form of reductions (binary, associative operations) performed on state vectors of values provided by the logical processes. Our hardware description is based on the completed design of a four-node prototype, designed and built at the University of Virginia.

## 1. INTRODUCTION

Despite the fact that parallel simulation and the potential speed-up it offers are regarded as crucial to many applications — digital network simulation, military simulations, air traffic control simulations, and the like — very few efforts have focussed on specialized hardware to support parallel simulation. Exceptions include Fujimoto's rollback chip (Fujimoto, Tsai, and Gopalakrishnan 1992) and virtual computer (Fujimoto 1989) efforts, a special-purpose global virtual time (GVT) network (Filoque, Gautrin, and Pottier 1991) and our own framework (Reynolds 1992). Recent simulation results (Srinivasan 1992) show beyond a doubt that specialized hardware such as the framework we describe here can yield significant benefits to parallel simulations.

We describe the details of a hardware realization of our framework (Reynolds 1992). This description is based on a completed design of a four-node prototype expected to be operational Summer, 1992. This prototype is designed to interface with a Sparc Cluster — a set of Sparc-1e engines connected through a VME backplane. The interface to the Sparcs is through the Sun SBus.

Goals of the design include speed, scalability, adaptability and generality. Simulations project our framework hardware will be able to compute GVT asynchronously in 10 to 20 microseconds on a 32-node system, easily two orders of magnitude faster than on conventional parallel processor host networks. While the prototype design includes a synchronous reduction network for speed, a scalable asynchronous design has been completed. We chose to go with the synchronous design in the prototype to reduce potential problems such as race conditions and to extract speed reliably. To achieve adaptability — the potential to interface to any parallel processor — we have isolated the SBus interface completely. Interconnection to other parallel processors or through alternate channels has been kept simple. We have kept the network general by incorporating high speed general purpose ALU's in the reduction network. All operations, including 32-bit fixed-point arithmetic, can be performed in less than 40 nanoseconds. The importance of attaining our goals will become clearer in the remainder of this paper.

In the sections that follow we present a brief overview of the framework first described in Reynolds (1991). The ease and speed with which important synchronization values such as Time Warp's GVT can be computed will be made evident. Also, we present a set of correctness criteria for the hardware portion of the framework. Race conditions were a persistent problem that had to be addressed from the outset. Following that we describe the hardware prototype in detail, and close with a discussion of the future directions of our effort.

## 2. MOTIVATION

Three components integral to the design of a low-level *framework* supporting parallel discrete event simulation (PDES) are: (1) small sets of global values needed by various PDES synchronization protocols, (2) a hardware-based method for rapidly reducing and disseminating these values, and (3) algorithms for using disseminated information to enhance efficiency in a

parallel simulation. Throughout this paper we use the term "framework" to refer to these three components. Sets of global values can be computed by performing reductions on input values across all processors. The high speed *framework hardware* (Reynolds and Pancerella 1992) rapidly computes and disseminates these values; the framework hardware we propose consists of a *parallel reduction network* (PRN) augmented with dedicated processors to manage the high frequency I/O from the network and execute synchronization algorithms. The algorithms calculate input values to the reduction operations and use the global output values to synchronize logical processes in any PDES. We discuss how rapidly reduced values can support various PDES synchronization protocols first. In the remainder of the paper we elaborate on the design of our framework hardware, the parallel reduction network and its interfaces.

Critical synchronization information can be disseminated in the form of reductions performed on values provided by logical processes (LP's). The underlying hardware computes and rapidly disseminates results of *global reduction operations*; reductions are binary, associative operations — for example, sum, minimum, maximum, logical AND, logical OR, etc. — performed on data across all processors. The success of disseminating global synchronization values is contingent on the high speed at which these values are made available to all processes. We expect a new global value to be emitted from the reduction network every 150 nanoseconds.

One such set of globally reduced values and related synchronization algorithms have been presented in detail in Reynolds (1992) and in Pancerella (1992). The values computed are the minimum next event time and the minimum logical timestamp of messages that have been sent but not acknowledged. These values, along with synchronization algorithms to correctly maintain them, are sufficient to eliminate causality errors and support deadlock-free parallel simulation even when message traffic is always present. The elimination of causality errors allows an LP to recognize when it can commit to processing an irreversible act such as I/O.

Simulations (Srinivasan 1992) show that messages can be acknowledged efficiently in a high speed reduction network (as proposed by Pancerella (1992)) in order to support the maintenance of a minimum outstanding message time. Message acknowledgements in a reduction network are supported by *tagged selective reduction operations*; in a selective reduction operation, such as minimum or maximum, a tag accompanies the "winning" value of the binary operation. The tag field for a message acknowledgement is a unique message ID.

In an aggressive PDES synchronization protocol (See Reynolds (1988)), such as Time Warp (Jefferson 1985), GVT can be efficiently computed by an LP *at any time* using our framework; it is simply, by definition, the minimum of the two globally reduced values: minimum next event time and minimum outstanding message time. GVT computation and dissemination in this hardware-based framework is a significant improvement in algorithm complexity and implementation efficiency over previously proposed GVT maintenance schemes (Jefferson 1985, Jefferson and Sowizral 1985, Samadi 1985, Lin and Lazowska 1989, Bellenot 1990, and Concepcion and Kelly 1991).

Researchers have shown that minimum event processing times and *lookahead* values can produce significant performance improvements in a non-aggressive PDES protocol (See Fujimoto (1987), Fujimoto (1988), Reed, Malony, and McCredie (1988), and Felderman and Kleinrock (1992)). A hardware-based framework can be used to calculate and disseminate the smallest future time that an LP can send event messages. Each LP computes the value it submits to this reduction based on its current local clock and its minimum processing time.

In addition to lookahead values, another possible enhancement to parallel simulations may be the rapid dissemination of reduction values which estimate the maximum (or minimum) rate at which an LP is processing events. If each LP submits a current estimate of its rate of simulation, the fastest (or slowest) LP (with respect to logical time) can be identified. Felderman and Kleinrock (1990) show analytically that a Time Warp simulation can be more efficient if a faster LP is slowed down; they do not propose how the information might be propagated. We have a hardware-based framework for disseminating this information easily.

Iterative PDES algorithms, such as Bounded Lag (Lubachevsky 1988), Moving Time Window (Sokol, Briscoe, and Wieland 1988), and the aggressive Global Windowing Algorithm discussed in Dickens and Reynolds (1992), require the rapid computation and dissemination of ceiling values or fault values. Lubachevsky has recognized that a special-purpose network can be used to broadcast a minimum event time in his bounded lag protocol (Lubachevsky 1988); a reduction network can efficiently support this PDES algorithm and other PDES windowing algorithms. The efficiency of these algorithms can be increased since these algorithms typically rely on a host network, much slower than our hardware, to disseminate windowing values. Furthermore, additional global reduction values, such as a minimum outstanding message time or a minimum next event time, could enhance iterative algorithms. For example, a window may be enlarged by including this additional knowledge.

Finally, the challenge of global termination detection and the calculation of output measures in a PDES (Abrams and Richardson 1991) can be realized easily within our framework. Many global termination conditions — for example, sums and boolean operations — can be calculated and disseminated efficiently in a reduction network. Unlike the distributed snapshot algorithm in Chandy and Lamport (1985), a framework consisting of synchronization values and related algorithms can be used to evaluate termination conditions even when there are outstanding messages in the parallel simulation. As mentioned above, computing minimum unreceived message times and acknowledging messages in a reduction network can be used in order to detect outstanding messages in a system. Moreover, a sum of the number of all messages sent minus messages received at all LP's can be computed in a reduction network to detect outstanding messages in the system. If this value is maintained correctly, a sum of zero indicates that there are no outstanding messages in the host communication network.

Our hardware-based framework supports the computation and dissemination of all of the values just discussed, across all processors without coordination of host processors, i.e., without barrier synchronization. The reduction network interfaces with dedicated auxiliary processors that manage the high speed I/O from the network. Employing auxiliary processors provides a separation of the synchronization activity (performed on auxiliary processors) and the application being simulated (performed on host processors). In sum, our framework offloads all parallel simulation synchronization overhead from host processors and the host network.

## 3. A TOP-LEVEL VIEW

Often sets of values such as those mentioned above have temporal relationships that must be preserved. In this section we discuss steps to ensure that necessary temporal relations can be maintained.

### 3.1. Correctness Criteria

There are two properties that must be ensured in the design of the framework hardware in order to maintain temporal relations among globally reduced values and, thus, to guarantee correctness of synchronization algorithms: (1) atomicity of read accesses to an instance of globally reduced values and (2) order preservation of inputs from a given LP. Meeting these requirements in hardware is challenging; hence, interfaces into and out of the reduction network must be designed with care.

When the PRN computes multiple reductions representing a state of a simulation, it is crucial that each LP can access a set of globally reduced values atomically to guarantee that the values represent a consistent global state. In addition to atomicity, some synchronization algorithms (Reynolds 1992) require that the order an LP changes values input to the network be preserved in global counterparts by the underlying hardware. This is necessary, for example, if an LP sends a message, updates a minimum unreceived message time, and then changes its current next event time, other LP's must see any effect that the unreceived message time has on its global minimum *no later than* the effect its new next event on the global minimum next event time is seen in the system. This ordering constraint prevents race conditions that can cause global reduction values to reflect an incorrect global state.

The *no later than* ordering property suggests that if two reduction input values, $v_{1_i}$ and $v_{2_i}$, are updated in order, this ordering must be guaranteed at two times: (1) when values enter the reduction network and (2) when their globally reduced counterparts leave it. There is no simple way to guarantee that an LP will see globally reduced values in exactly the same order in which they had local changes submitted to them on an input side of the network. Even if the network is designed to input local values from an $LP_i$ in the order in which $LP_i$ changes them, and therefore emit any changes to globally reduced values in the same order, when the network emits these values, there is no simple way to guarantee that they will be processed in the same order. 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 sequence of events in the following example.

Suppose $LP_i$ changes $v_{1_i}$, which is then read by the reduction network. Next, $LP_i$ changes $v_{2_i}$, which is then read by the reduction network. A second logical process, $LP_j$, reads the old globally reduced $v_1'$, i.e., without the impact of $v_{1_i}$. The reduction network first emits the new $v_1'$, i.e. with impact of $v_{1_i}$, and then emits the new $v_2'$, i.e. with impact of $v_{2_i}$. Finally, $LP_j$ reads new $v_2'$.

Note, with these events, that $LP_j$ is reading reduced 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 $v_{1_i}$ is always changed before $v_{2_i}$. We conclude the hardware should compute different binary, associative operations across *state vectors* of values. Each element of a state vector is an input to a binary, associative operation. The state vector is the basic unit of operation in an implementation of the framework hardware: the hardware reads a state vector of size $m$, computes $m$ globally reduced values, and writes a globally reduced state vector. The hardware must guarantee that a partial or incomplete global state vector is never seen by the application. State vectors are sufficient to ensure the *no later than* property; they keep logically related data together, capture consistent

snapshots of the application, and avoid race conditions. Application programs, in turn, must then guarantee that state vectors which are fed into the reduction network represent valid states.

There are two interesting ways in which an auxiliary processor may modify its local state vector. It is desirable that the hardware support these cases:

1) *atomic write without overwrite* — one or more of the 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 state vector; and
2) *atomic write with overwrite* — one or more of the local values are changed effectively simultaneously, yet further changes to the local state vector may be made prior to the network reading the state vector.

We now discuss a hardware design for a parallel reduction network which supports PDES in a manner consistent with our established correctness criteria.

### 3.2. A Functional View of the Framework Hardware

A top-level view of the system including the framework hardware appears in Figure 1. The host system for the framework hardware is a closely coupled network of high speed processors with its own network for interprocess communication. This host network is independent of the framework hardware. In our prototype the host system is a Sparc Cluster where Sparcs can communicate through a VME backplane.

Each *host processor* (HP) is paired with a corresponding *auxiliary processor* (AP) which interfaces directly to the reduction network. The general-purpose auxiliary processors, one processor per host processor, provide the interface between host processors and the reduction network. There is a high speed bidirectional communication channel between a host processor and its corresponding auxiliary processor. The prototype interface between a host, a Sparc-1e, and the 32-bit general purpose auxiliary processor, a 25 MHz Motorola 68020, is a dual-ported RAM connecting the Sun SBus and the auxiliary processor. The SBus has a bandwidth of about 100 megabytes per second for 32-bit words (Sun Microsystems 1990). We expect a potential throughput of 25 megabytes per second from host to auxiliary processor.

Each AP has 256 Kbytes of RAM — expandable up to 1Mbyte — to store synchronization programs and related data structures (See Reynolds (1992), Pancerella (1992), and Srinivasan (1992)). Furthermore, each AP has 128 Kbytes of EPROM to store a boot-up monitor which is executed upon reset.

The addition of auxiliary processors at the interface to the reduction network facilitates the management of high-frequency data coming out of the network. We estimate a new state vector to be written every $150 * m$ nanoseconds, where $m$ is the number of elements in a state vector. The AP's are responsible for inserting state vectors into the PRN as well but with much lower frequency since this is performed under program control.

High speed synchronization activity in the parallel simulation framework is performed on the dedicated AP's. Host processors are responsible for executing events and sending/receiving event messages, and auxiliary processors are responsible for executing framework synchronization algorithms (See Pancerella (1992) and Srinivasan (1992)). When an AP reads new globally reduced values from the network, it writes selected groups of these values into the HP-AP interface readable by the host processor. An LP, executing on a host processor, can compute GVT, avoid deadlocks, and make processing decisions based on the global synchronization values. Other than simple tests such as these, the execution of the framework algorithms does not interfere with an LP's event processing. A further advantage of a dedicated processor interfacing with the host processor and the reduction network is that an AP can compute the input reduction values based on multiple LP's executing on one host processor and coordinate the synchronization activity of multiple LP's.

The specific details of the interfaces — both between a host processor and its auxiliary processor and between an auxiliary processor and the PRN (both input and output) — are discussed in later sections.

## 4. DETAILS OF THE HARDWARE DESIGN

In the sections that follow we discuss the specifics of the hardware in our prototype design. In this discussion we focus on how we ensure the correctness criteria established earlier.

### 4.1. Setup

Each auxiliary processor boots up in a "listening" state in which it monitors its host processor interface. A host processor sends tagged data to its auxiliary processor representing a program to be loaded and executed by the AP. The physical interface between a host processor and its auxiliary processor is described in the next section.

One of the host processors in the system and its corresponding auxiliary processor is designated as a *master pair* of processors. The master pair communicates PRN programming information to the state machine controlling 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 is to be taken on all third components in a three component state vector.

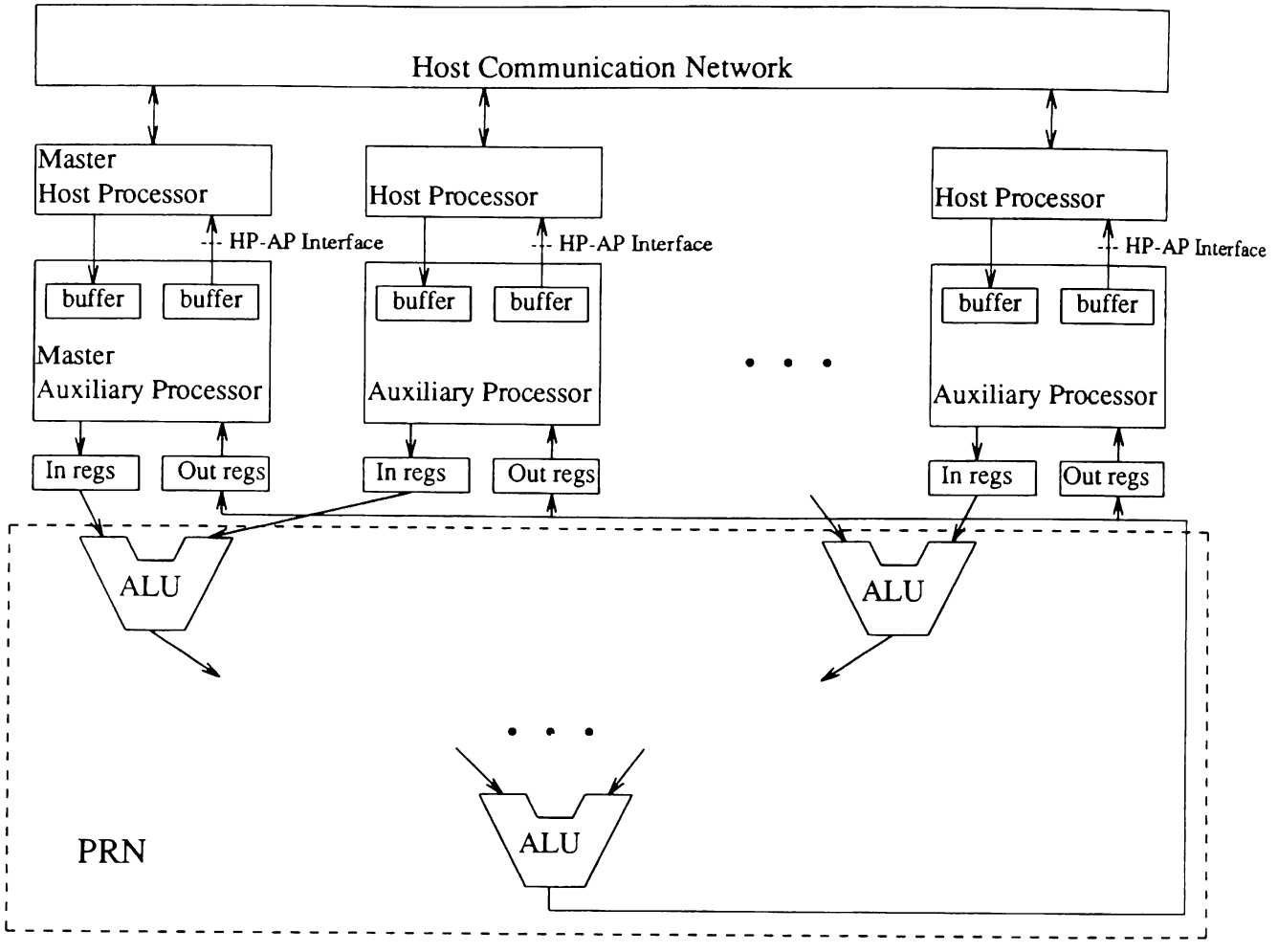


**Figure 1: High Level View of Framework Hardware**

The master host processor can send tagged data representing new PRN programming information to its auxiliary processor at any time. Similarly, host processors can send data to their respective auxiliary processors indicating 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.

### 4.2. Host Processor - Auxiliary Processor Interface

Functionally, there are two data paths between a host processor and auxiliary processor: one from the HP to the AP and the other from the AP to the HP. Each processor is a reader on one data path and a writer on the other path. The host occasionally writes tagged information to the interface which the AP processes, based on the tag, and generates values to input into the PRN. Similarly, the AP writes globally reduced values to the interface which is read by the HP. Framework algorithms require that (1) no information sent by the HP is lost and (2) the AP processes the data in the order in which it is sent by the HP. Under the established correctness criteria, an application executing on the HP does not need to see all globally reduced values; a recent version of globally reduced values, however, is expected to be available to the HP. Hence, the implementation requires at least a FIFO queue from HP to AP and a single set of registers from AP to HP.

The prototype interface between HP and AP is implemented by a dual-ported RAM, such that the host

processor is connected to one port and the auxiliary processor is connected to the other. Each of these ports is memory-mapped into the respective processor's address space. The two data paths are managed in the dual-ported RAM by software resident on the host and auxiliary processors; soft semaphores rely on the exclusive-write support provided by the dual-ported RAM.

The host processor accesses the dual-ported RAM via SBus. This HP interface isolates the particular host processor — a Sparc-1e in the prototype — from the rest of the system. If the host system changes, this HP interface is the only thing that will need to be redesigned. Isolating the HP interface provides adaptability to other parallel processors or closely coupled networks. For example, the SBus interface could be changed to a SCSI or VME interface, and all that would be required is the logic to respond to requests by the HP on the dual-ported memory.

### 4.3. The Parallel Reduction Network

As seen in Figure 1, the PRN is a binary tree of depth $\log_2 n$, where $n$ is the number of host (and auxiliary) processors. Each stage of the PRN consists of half as many ALU's as the stage above it, with the first stage having $n/2$ ALU's. The PRN's binary tree properties allow a global reduction operation to be computed and disseminated in $O(\log n)$ time.

A single ALU node is shown in Figure 2. The ALU's in the prototype parallel reduction network require 40 nanoseconds to perform a 32-bit fixed-point addition. Each 32-bit input register is paired with a 32-bit tag register. The PRN propagates the tag of the input that "wins" a selective operation, a minimum or maximum operation, so that the tag of the smallest or largest element emerges from the bottom of the PRN for a minimum or maximum operation. In the case where there is no single choice in a selective operation (i.e., both operands are equal), the PRN selects deterministically the tag which is propagated. A selective operation requires two operations in the ALU: a compare and a select; hence, this requires 80 nanoseconds in our prototype.

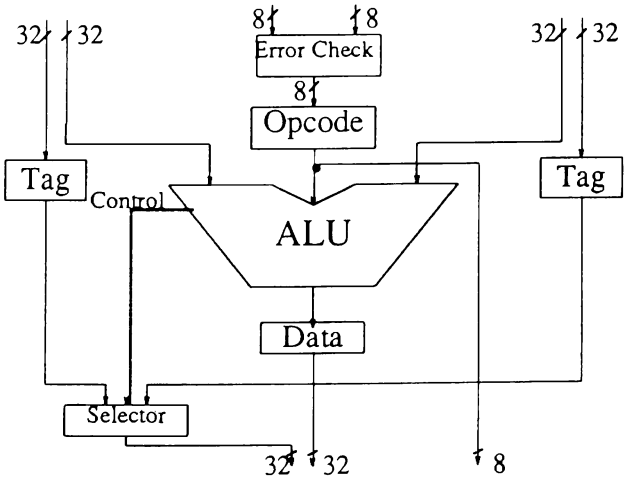


**Figure 2: An ALU Node in the PRN**

Pipelining is employed in order to use this network efficiently: partial results are pipelined through the $\log n$ stages of the PRN such that each stage of ALU's is always busy. The PRN can pipeline binary, associative operations at a rate equal to the delay time of a stage. The time for a value to pass from one level of the PRN to the next is a *minor cycle*. Currently, this delay is projected to be no more than 150 nanoseconds. Thus, the time to produce a sequence of values for state vectors of length $m$ is 150 * $m$ nanoseconds (plus the time to fill the pipe: 150 * $\log_2 n$ nanoseconds).

The pipelining in the prototype is performed synchronously. An asynchronous design for the reduction network has been completed. In the asynchronous design, each ALU node of the PRN computes and outputs a result once it has completed an operation and two input values are available from the preceding stage. Each PRN node operates in a demand-driven manner, where operations are performed as both inputs become available. This asynchronous design is desirable for later versions of the reduction network for two reasons. First, a PRN operating asynchronously is scalable since a hardware handshake can be used to control communication between nodes; this eliminates both a central clock in the PRN and the potential problem of clock skew in a large network. Second, this facilitates the addition of floating point processors at each ALU node. A long operation, such as a floating point operation, forms a one-time "bubble" in the pipeline. With a synchronous network, the minor cycle must allow for the longest operation. Thus, a synchronous design creates wasted time when a shorter binary, associative operation is performed, and an asynchronous design alleviates this problem. We note that the synchronous design is simpler, and it is faster when only operations with uniform execution times are performed.

As seen in Figure 1, the interface to the PRN from each processor is identical. Each AP 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 globally reduced results into the output registers. This memory-mapped interface is a possible source of memory contention if both the PRN and the

auxiliary processor attempt to access the input or output registers simultaneously. We discuss next how the interface between the processor and the PRN is constructed in order to minimize the memory contention, to facilitate atomic writes with and without overwrite capabilities, and to preserve state vectors.

### 4.4. Auxiliary Processor-PRN Interface

The AP-PRN interface is designed to operate on *state vectors* in order to support both atomic accesses of globally reduced values and order preservation of input values to the reduction network. From an LP's point of view, it feeds a *valid* state vector to the PRN, where "valid" is defined by the application using the framework hardware. Furthermore, the hardware provides an atomic read access to a single output state vector so than an AP can read an entire state vector. The application software, however, must access whole state vectors, not individual elements.

The prototype hardware limits state vectors to size eight; each of the eight elements is a register pair, one 32-bit data register and one 32-bit tag register. The data register can be a 32-bit integer, a 32-bit fixed point number, or any 32-bit logical value, depending on the reduction operation to be applied. All numeric values are two's complement. The tag register can contain any 32-bit value. The PRN can be programmed to operate on state vectors of size two to eight, depending on the application. The PRN reads the state vectors, processes them by performing the corresponding reduction on each element, and writes a globally reduced state vector at each AP.

An auxiliary processor and the reduction network operate asynchronously with respect to one another. As shown in Figure 3, three banks of eight input and output register pairs provide an interface of isolation, such that both can access the register banks with minimal interference. This interface is designed to guarantee that *the PRN never blocks* while waiting to read a value or write a value. The PRN is expected to read and process state vectors at a rate much faster than an AP produces them; the PRN, therefore, may read and process the same state vector repeatedly. Similarly, on the output side, the PRN will produce globally reduced state vectors faster than an AP can read and process them, and as a result the AP's may lose some state vectors. All reads to registers from the PRN or an AP are nondestructive. We now discuss the input and output interfaces in greater detail.

#### 4.4.1. Auxiliary Processor-PRN Interface: Input

The interface from an auxiliary processor to the PRN consists of three banks of eight register pairs: the *AP input registers*, the *Intermediate input registers*, and the *PRN input registers*. The AP writes state vectors of size $m$, where $m$ is between two and eight, to the top row of registers, the AP input registers, and the PRN reads state vectors of size $m$ from the bottom row, the PRN input registers. The state machine which controls the interface transfers state vectors from the AP input registers to the Intermediate input registers and then to the PRN input registers. The transfer is done so as to minimize interference. Intermediate registers facilitate getting snapshots of valid local state vectors to be passed on to the PRN input registers without blocking the PRN.

When an auxiliary processor has completed writing a new state vector, it sets two single-bit control flags: the *overwrite bit* (OW) and the *owner bit* (O). The owner bit is *always* set when the AP has finished writing a valid state vector into the AP input registers; this indicates that the interface controller now owns the top level of registers. When the interface state machine transfers this state vector to the Intermediate input registers, it resets the owner bit indicating that the AP once again owns the AP input registers. If the AP attempts to write to the AP input registers while the owner bit is still set, it will be blocked. However, given the relative speeds of the PRN and the AP, this is not expected to happen often.

The overwrite bit gives the application some control over what values are eventually fed into the reduction network. Specifically, if the AP marks a state vector as "non-overwritable", it is guaranteed that the entire vector will be processed by the PRN. When the control logic transfers the AP input registers to the intermediate level, the overwrite bit is also transferred. If the AP indicates a state vector is overwritable then the state machine controlling the register banks can allow subsequent state vectors written by the AP to overwrite the state vector in the Intermediate input registers. If the AP signals a state vector as non-overwritable and it is transferred to the Intermediate registers, the overwrite bit will prevent the transfer of a newly written AP level state vector until the Intermediate input registers are ultimately transferred to the PRN input registers. The control logic guarantees that AP input registers are only moved to the Intermediate input registers when this process does not cause the PRN to block or when it does not lead to a loss of integrity of a state vector. Finally, we note that due to the relative speeds of an AP and the PRN, it is very unlikely that an overwritable state vector will be overwritten prior to being read by the PRN; however, we have designed the network to provide the guarantee anyway, for future use.

The PRN reads state vectors of a specified size cyclically, starting with the $m$th element and proceeding to the first element. Thus, the PRN reduces the $m$th element, followed by the $(m-1)$st, and so on. The PRN is pipelined, thus the processing of the $(i-1)$st elements commences as soon as the top level of ALU's completes processing the $i$th elements. The PRN reads the $i$th register pair from each of the $n$ input banks

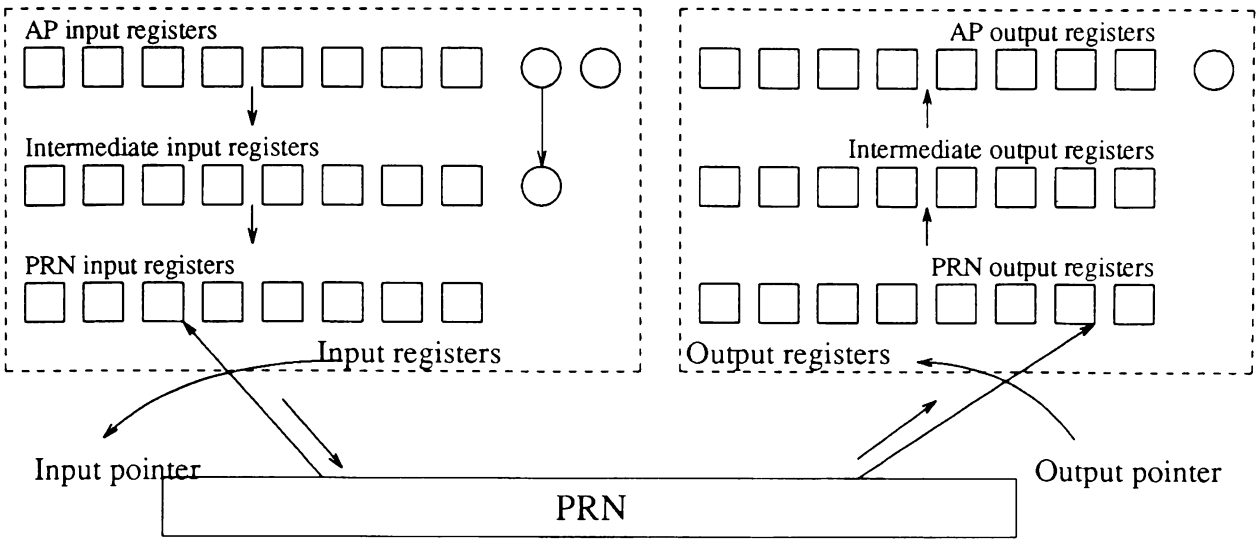


**Figure 3: Interface Between an Auxiliary Processor and the PRN**

simultaneously. The time for the PRN to read an entire state vector is an *input cycle*. An input cycle finishes when the first elements of the state vector are consumed. At the end of an input cycle, the controller transfers the Intermediate input registers to the PRN input registers. The transfer is overlapped with the last PRN read in the input cycle; for this reason, our hardware requires a minimum state vector size of two. The transfer from the Intermediate registers to the PRN registers has a higher priority than the transfer from the AP registers to the Intermediate registers so that the PRN never blocks.

We note that $\log_2$ of $n$ and $m$ are not necessarily equal. Therefore, while the PRN is reading from the $i$th input register pair from all $n$ processors, it is not necessarily writing the $i$th output register pairs. That is, the PRN may complete reading state vectors from each of $n$ input register banks at a different time than when it completes writing new reduced state vectors. The writing of a reduced state vector for a set of input state vectors will lag by $(((m-1) + \log_2 n) * 150)$ nanoseconds, where the minor cycle time is 150 nanoseconds and there are $n$ processors.

### 4.4.2. Auxiliary Processor-PRN Interface: Output

As shown in Figure 3, the three banks of output registers are constructed to preserve state vectors and to minimize AP-PRN interference in a similar fashion to the input register banks. Once every $m$ minor cycles (assuming a full pipe in the PRN), the PRN generates a globally reduced state vector, which is written to the *PRN output registers*. This state vector is transferred to the *Intermediate output registers* and finally to the *AP output registers*, which are readable by the AP. Once again the interface controller guarantees that the PRN never blocks, and transfers between output register levels are prioritized to prevent this.

Each time the PRN completes writing a state vector into the bottom row of registers, the values are shifted into the Intermediate output registers. When the bottom row is shifted, the values in the intermediate row are concurrently shifted into the AP output registers unless the AP has locked the top row because it is reading the AP output registers. In that event, the Intermediate output registers are overwritten by the PRN output registers, and the contents in the intermediate registers are lost forever. The AP output registers have a control bit, an owner bit (O), that is set and reset by the auxiliary processor. The owner bit determines whether Intermediate output registers can be written to the AP output registers or are lost; it also ensures an atomic read of a state vector by the AP. The AP may block momentarily if it attempts to set the owner bit to itself while the intermediate values are written in parallel to the registers readable by the AP. Applications using the framework hardware must be robust enough to tolerate the loss of state vectors emerging from the PRN. We note that an AP never sees a partial state vector. State vectors are either lost in their entirety or not at all.

If an application cannot tolerate state vector loss of globally reduced states, an alternative is to use two extra input registers and compute tagged selective operations to perform a double handshake (Pancerella 1992). In this scheme, all updated globally reduced state vectors can be read by the auxiliary processors even though physical state vectors may still be lost. We note, however, that it is expensive (in terms of computation time) to implement this property in the framework hardware and it should be avoided when possible.

## 5. RELATED WORK

Using a separate synchronization network for improving system performance is not a new idea. The IBM RP3 (Pfister, *et. al.* 1985) 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 reduction network is not as complex or expensive as a combining network, yet it performs global synchronization operations very efficiently.

We claim no novelty with respect to reduction networks. Lubachevsky (1988) 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 this PDES protocol. The Finite Element Machine (FEM) (Jordan, Scalabrin, and Calvert 1979 and Crockett and Knott 1985), 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. 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 set of input and output registers which are treated as a single state vector, whereas the FEM uses a single input and a single output register.

At about the same time that we introduced our framework, Filoque, Gautrin, and Pottier (1991) 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 Filoque, Gautrin, and Pottier (1991) rely on the host communication network for message acknowledgements and our framework uses the framework hardware for this purpose. The goals of both approaches are similar, but our framework is more efficient, more flexible, and more scalable.

Several researchers have proposed the use of hardware to implement barrier synchronization. Hoshino (1985) has an efficient barrier synchronization in the PAX computer. Stone (1990) suggests the use of global busses to compute maximum values and to implement fetch-and-increment. The hardware that we propose, on the other hand, provides support for a larger class of algorithms than barrier synchronization algorithms.

Many parallel architectures provide for global binary, associative operations across all processors. Global operations on the Intel iPSC/2 (Intel Corporation 1989) are provided for arithmetic and logical operations. The CM-5 (Thinking Machines Corporation 1992) 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 CM-5 control network supports "soft" barrier synchronization, arithmetic and logical reduction operations, parallel prefix operations, and segmented parallel prefix operations. The reduction operations on both of these machines require the complete synchronization of all processors. All processors must call global operation functions with a contributed value, and a global operation blocks until all processors enter it. Our framework hardware, on the other hand, computes and disseminates globally reduced values on state vectors *without* the coordination of the host processors; the reduction operations on the PRN are performed continuously (i.e., allowing stale data to be contributed to global operations). Furthermore, the hardware design employs auxiliary processors to manage the high-speed data emitted from the reduction network.

## 6. CONCLUSIONS

We have introduced prototype framework hardware, a parallel reduction network augmented with dedicated processors, as special-purpose hardware to rapidly compute and disseminate synchronization information, based on state vectors, in a parallel simulation. This hardware supports an experimental approach to PDES; we can easily change the critical state information that is disseminated and the related synchronization algorithms in order to study various PDES protocols. We expect new reduction values and algorithms within the framework to develop over time. Our longer term goals lie in demonstrating how this hardware-based framework can support adaptive

protocols which have been neglected because of the costs associated with gathering and using information for adaptive decisions. It has been suggested that an adaptive protocol, one which is an intelligent combination of aggressive and non-aggressive protocols, may be more powerful than either in a pure form (Reynolds 1988, Reynolds 1992, and Lubachevsky, Weiss, and Shwartz 1991).

In the future we intend to report on the performance of the prototype PRN, which is a collaborative effort of the Computer Science and Electrical Engineering Departments at the University of Virginia to be completed in Summer 1992. Simulations (Srinivasan 1992) have demonstrated the feasibility of this hardware; however, actual PDES implementations on the prototype will validate our performance predictions. Furthermore, we have proposed to integrate our hardware, which offloads global synchronization in a PDES, with Fujimoto's rollback chip (Fujimoto, Tsai, and Gopalakrishnan 1992), which offloads state saving and state restoration in an aggressive PDES, in order to build a Parallel Simulation Engine.

Finally, an open research problem is the construction of the next version of the framework hardware which rapidly disseminates *target-specific* reductions on contributed values. A PDES with completely static properties — the number of LP's and the communication topology are known *a priori* — should show significant runtime speedup if the framework hardware were enhanced to include the dissemination of target-specific synchronization information, where an LP receives reduced values from its predecessors as determined by the transitive closure of the static communication graph. Hence, each LP receives reduced information only from those LP's that can have an impact on its performance. By providing efficient dissemination of target-specific synchronization information to all LP's in a PDES, the LP's receive more accurate state information and can make event processing decisions accordingly. The final result would be a hardware-based framework for PDES that efficiently supports a wide range of parallel simulations.

## ACKNOWLEDGEMENTS

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

## REFERENCES

Abrams, M. and Richardson, D. 1991. Implementing a Global Termination Condition and Collecting Output Measures in Parallel Simulation. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, 86-91. Anaheim, California.

Bellenot, S. 1990. Global Virtual Time Algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 122-127. San Diego, California.

Chandy, K. M. and Lamport, L. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM TOCS* 3(1):63-75.

Concepcion, A. I. and Kelly, S. G. 1991. Computing Global Virtual Time Using the Multi-Level Token Passing Algorithm. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, 63-68. Anaheim, California.

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

Dickens, P. M. and Reynolds Jr., P. F. 1992. State Saving and Rollback Costs for an Aggressive Global Windowing Algorithm. TR-92-18, Dept. of Computer Science, University of Virginia, Charlottesville, Virginia.

Felderman, R. E. and Kleinrock, L. 1990. Two Processor Time Warp Analysis: Capturing the Effects of Message Queueing and Rollback/State Saving Costs. Submitted to *ACM TOMACS*.

Felderman, R. E. and Kleinrock, L. 1992. Two Processor Conservative Simulation Analysis. In *Proceedings of the 1992 Western Simulation MultiConference on Parallel and Distributed Simulation*, 169-177. Newport Beach, California.

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

Fujimoto, R. M. 1987. Performance Measurements of Distributed Simulation Strategies. TR UUCS-87-026a, Computer Science Dept., University of Utah, Salt Lake City, Utah.

Fujimoto, R. M. 1988. Lookahead in Parallel Discrete Event Simulation. In *Proceedings of the 1988 International Conference on Parallel Processing*, 34-41. University Park, Pennsylvania.

Fujimoto, R. M. 1989. The Virtual Time Machine. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, 199-208. Santa Fe, New Mexico.

Fujimoto, R. M., Tsai, J. J. and Gopalakrishnan, G. C. 1992. Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp. *IEEE Trans. on Computers* 41(1):68-82.

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

Intel Corporation. 1989. **iPSC/2 Programmer's Reference Manual**. Beaverton, Oregon: Intel Scientific Computers.

Jefferson, D. R. 1985. Virtual Time. *ACM TOPLAS* 7(3): 404-425.

Jefferson, D. and Sowizral, H. 1985. Fast Concurrent Simulation Using the Time Warp Mechanism. In *Proceedings of the Conference on Distributed Simulation*, 63-69. San Diego, California.

Jordan, H. F., Scalabrin, M. and Calvert, W. 1979. A Comparison of Three Types of Multiprocessor Algorithms. In *Proceedings of the 1979 International Conference on Parallel Processing*, 231-238.

Lin, Y. B. and Lazowska, E. D. 1989. Determining the Global Virtual Time in a Distributed Simulation. TR 90-01-02, Dept. of Computer Science, University of Washington, Seattle, Washington.

Lubachevsky, B. D. 1988. Bounded Lag Distributed Discrete Event Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 183-191. San Diego, California.

Lubachevsky, B., Weiss, A. and Shwartz, A. 1991. An Analysis of Rollback-Based Simulation. *ACM TOMACS* 1(2):154-193.

Pancerella, C. M. 1992. Improving the Efficiency of a Framework for Parallel Simulations. In *Proceedings of the 1992 Western Simulation MultiConference on Parallel and Distributed Simulation*, 22-29. Newport Beach, California.

Pfister, G. F., Brantley, W. C., George, D. A., *et. al.*. 1985. The IBM Research Parallel Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, 764-771. St. Charles, Illinois.

Reed, D. A., Malony, A. D. and McCredie, B. D. 1988. Parallel Discrete Event Simulation Using Shared Memory. *IEEE Trans. on Software Engineering* 14(4):541-553.

Reynolds Jr., P. F. 1988. A Spectrum of Options for Parallel Simulation. In *Proceedings of the 1988 Winter Simulation Conference*, 325-332. San Diego, California.

Reynolds Jr., P. F. 1991. An Efficient Framework for Parallel Simulations. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, 167-174. Anaheim, California.

Reynolds Jr., P. F. and Pancerella, C. M. 1992. Hardware Support for Parallel Discrete Event Simulations. TR-92-08, Dept. of Computer Science, University of Virginia, Charlottesville, Virginia.

Reynolds Jr., P. F. 1992. An Efficient Framework for Parallel Simulations. To appear in *International Journal on Computer Simulations*.

Samadi, B. 1985. Distributed Simulation, Algorithms, and Performance Analysis. PhD Thesis, Computer Science Dept., UCLA, Los Angeles, California.

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

Srinivasan, S. 1992. Modeling a Framework for Parallel Simulations. Master's Thesis, Dept. of Computer Science, University of Virginia, Charlottesville, Virginia.

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

Sun Microsystems. 1990. **SBus Specification B.0**. Mountain View, California: Sun Microsystems, Inc.

Thinking Machines Corporation. 1992. **The Connection Machine CM-5 Technical Summary**. Cambridge, Massachusetts: Thinking Machines Corporation.

## AUTHOR BIOGRAPHIES

**PAUL F. REYNOLDS, JR.**, Ph.D., University of Texas at Austin, '79, is an Associate Professor of Computer Science at the University of Virginia. He has been a member of the faculty at UVa since 1980. He has published widely in the area of parallel computation, specifically in parallel simulation, and parallel language and algorithm design. He has served on a number of national committees and advisory groups as an expert on parallel computation, and more specifically as an expert on parallel simulation. He has been a consultant to numerous corporations and government agencies in the systems and simulation areas.

**CARMEN M. PANCERELLA** is a Ph.D. student at the University of Virginia, where she received a M.S. in Computer Science in 1989. She received a B.S. in Computer Science from Wilkes College in 1986. She is a student member of both the ACM and the IEEE Computer Society. Her research interests include parallel simulation and hardware design for parallel computing.

**SUDHIR SRINIVASAN** is a Ph.D. student at the University of Virginia. He received a M.S. in Computer Science from the University of Virginia in 1992 and a B.E. in Computer Science from Bangalore University, Bangalore, India, in 1990. His current research interest is parallel simulation. His other interests include parallel and distributed computing, compilers, parallel algorithms and parallel programming. He is a student member of the ACM.