Non-interfering GVT Computation via Asynchronous Global Reductions

Sudhir Srinivasan
Paul F. Reynolds, Jr.

Computer Science Report No. TR-93-17
April 1, 1993

Non-interfering GVT Computation via Asynchronous Global Reductions*

GVT computation has been a significant problem in Time Warp implementations since it affects the completion time of the simulation. We present a new method for computing GVT using inexpensive hardware designed to support synchronization in parallel computations. The main contribution of this paper is this new GVT algorithm. Unlike previous methods, the computation of GVT using our method does not interfere with the simulation. For the LP's, the cost of obtaining GVT is reduced to insignificance. Preliminary studies indicate that our method provides very accurate GVT.

Keywords: Modeling methodology - parallel and distributed simulation, PDES protocols, Time Warp, asynchronous GVT computation, hardware support.

* Published in the proceedings of the 1993 Winter Simulation Conference, Los Angeles.

1 Introduction

Since the introduction of Virtual Time and the Time Warp protocol [6], several issues have arisen regarding the performance of this protocol. One of these is the computation of global virtual time (GVT). GVT is an important value required by the Time Warp protocol. The method used to compute GVT can significantly impact the completion time of the simulation. It may even cause processes to block, which may, under certain conditions, lead to deadlock. In the past, several algorithms have been proposed to efficiently compute GVT. Two criteria for the “goodness” of a GVT computing algorithm have been (i) the number of messages required to compute GVT and (ii) the accuracy of the computed GVT (i.e., how close the computed value is to the actual value). We present a new, very efficient GVT computing algorithm which is based on inexpensive hardware designed to support synchronization in parallel computations in general and parallel discrete event simulations in particular.

Parallel discrete event simulation (PDES) concerns itself with the execution of a discrete-event simulation on multiple processors with the goal of reducing the completion time of the simulation. While there seems to be substantial concurrency in typical simulations, this concurrency has proven to be difficult to extract in practice. PDES exhibits what is commonly called *control parallelism* and consequently, well known techniques which have been used to extract data parallelism are ineffective. The common approach to PDES is to partition the simulated system into multiple components each of which is simulated by a separate discrete-event simulator called a *logical process* (LP). These LP’s are synchronized using a *PDES protocol*. Several protocols have been proposed in the literature [5] each having its advantages and drawbacks.

Time Warp [6] is a protocol in which the LP’s simulate events aggressively (i.e. which allows for causality errors). Consequently, they must periodically save state so that they may roll back to a correct state upon a causality error. These two operations have associated overheads which can degrade the performance of a Time Warp simulation. Thus, Time Warp performs well when the simulation exhibits “temporal locality”.

1.1 Importance of GVT

Another feature of Time Warp which introduces overheads is the computation of global virtual time (GVT). GVT represents the simulated time up to which all LP’s have simulated correctly and beyond which, all LP’s have simulated speculatively (i.e., may be incorrect). This leads to the property that no LP can ever be rolled back to a time earlier than the value of GVT. Thus GVT defines a commitment horizon, i.e., when GVT is greater than or equal to the timestamp of an event, the event cannot be rolled back again and therefore may be committed. Therefore, GVT is critical in interactive and real-time simulations. GVT is also used for several other purposes in Time Warp. The most important of these is fossil collection. Since LP’s have to save state periodically, they consume memory. However, all states with timestamps smaller than GVT may be reclaimed since they are “unreachable”. Other uses of GVT include termination detection and gathering statistics. Since computing GVT corresponds to obtaining a distributed global snapshot of the system, it is a non-trivial problem.

1.2 Previous GVT algorithms

In the past, several algorithms have been proposed to compute GVT in a message passing system [1, 2, 3, 4, 7, 8, 9, 11, 16, 19]. All of these algorithms use the host communication network in some way to compute GVT. The earlier algorithms [2, 11, 16] require each message to be acknowledged. While Lin and Lazowska [7] eliminate acknowledgments, they introduce a worst case of O(n^2) control messages. Concepcion and Kelly [3] proposed an algorithm in which dedicated processors are used to compute GVT but which requires acknowledgments as well as two messages for every Time Warp message. Bauer and

Sporrer [1] eliminate acknowledgments but the accuracy of their algorithm depends on how often processors report their state to a central GVT calculator. Mattern's algorithm [9] accounts for messages in transit by waiting until they are all received. In addition, all of the above algorithms use the host network to disseminate the computed GVT to the processors. Tomlinson and Garg [19] describe a method to detect when GVT has exceeded a bound rather than approximate a value of GVT. Filloque, Gautrin and Pottier [4] and Livny and Manber [8] use special hardware to compute GVT. However, they use the host network to acknowledge messages. The uniqueness of our algorithm is that it does not use the host communication network in any way.

1.3 Effects of GVT computation

From the description in Section 1.2, we see that a common feature of all previous GVT algorithms is that they use the host communication network either directly or indirectly to compute GVT. Inter-processor communication in current multi-computers is still relatively expensive. Thus the GVT computation interferes with the simulation in the following ways:

- It generates extra messages in the interconnection network. This increase in message traffic increases the average transmission time of messages due to congestion, which indirectly slows down the simulation.

- In some GVT computation algorithms, the LP's stop simulating in order to compute GVT. Obviously, this slows down the simulation.

- The algorithms are time-consuming (at least on the order of milliseconds per value of GVT) and consequently provide relatively stale values of GVT. This has two effects on the simulation: (i) Fossils cannot be quickly identified and collected. Therefore, LP's may block due to insufficient memory which slows down the simulation unnecessarily. In addition, if GVT is not computed independently of the simulation, this blocking introduces the potential for deadlock. (ii) Since fossil collection is costly, state is not saved very frequently in an effort to reduce memory usage. As a result, LP's generally have to roll back farther than required and rebuild state, which increases the rollback overhead and therefore the completion time [12].

The main focus of research in this area has been to minimize the number of messages used to compute a value of GVT while keeping the computed value of GVT as close to the actual value of GVT as possible. In this respect, our algorithm is most efficient, as it simply does not use the host communication network to compute GVT. Instead, GVT is computed asynchronously using hardware described later. Even though the algorithm requires messages to be acknowledged, acknowledgments are performed using the hardware. A second advantage of our algorithm is that since GVT is computed asynchronously, all LP's need not be involved in every GVT computation. An LP may obtain the value of GVT simply by reading a memory location. Finally, simulations indicate that the GVT thus obtained will be very accurate (few tens of microseconds old).

In section 2 we describe briefly the hardware used by the algorithm. The GVT computation algorithm is presented in section 3. In section 4 we discuss the performance of the algorithm. Section 5 concludes the paper.

2 Hardware support for PDES

In this section, we describe briefly the design and operation of the hardware used by our GVT algorithm. A detailed description of the hardware may be found in [15]. The hardware was designed as a part of a universal framework proposed in [14] to support all PDES's. Henceforth, we will refer to this hardware

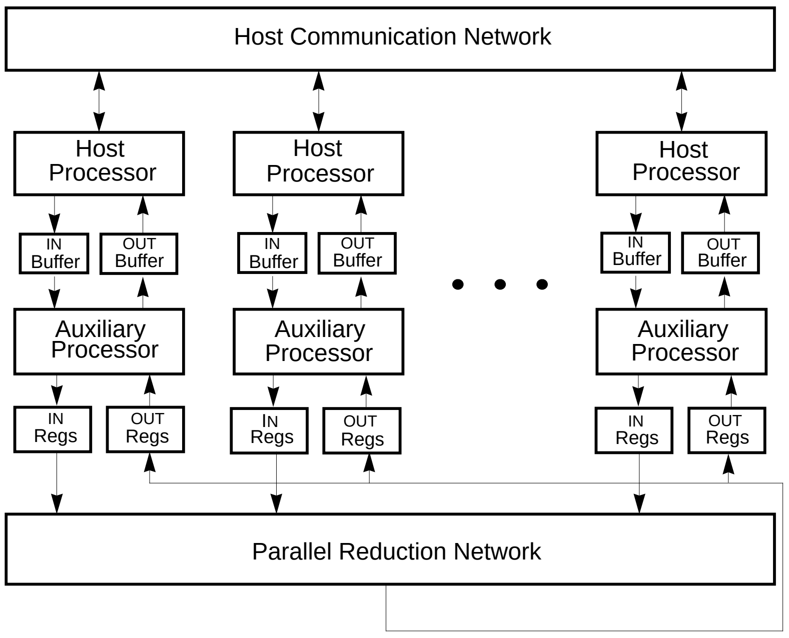

Figure 1 - Hardware Configuration

as the *framework hardware* or simply the *hardware*. Some of the features of the design of the framework hardware which enable it to support synchronization in general parallel computations are:

- speed - the hardware is designed to compute and disseminate global synchronization information very rapidly (on the order of hundreds of nanoseconds).

- scalability - the processing time of the hardware increases logarithmically with the number of processors while the number of components in the hardware increases linearly.

- adaptability - the design includes an interface to the host computing system which serves to isolate the design of the rest of the framework hardware from the host computing system.

- generality - the framework hardware contains programmable ALU's which allow it to be used to support a wide variety of applications.

- low cost - a prototype system for four processors has been built for twenty thousand dollars; a production system is expected to cost much less.

Thus, it must be noted that we are not proposing the design of special hardware to compute GVT. Instead, we are illustrating the use of this general synchronization hardware to compute GVT.

The hardware configuration is shown in Figure 1. The *parallel reduction network* (PRN) forms the core of the framework hardware. The PRN is a binary tree of height $\log_2 n$, where n is the number of processors. Figure 2 shows a PRN for eight processors. Each node of the tree contains a general purpose ALU which performs binary associative reduction operations (such as MIN, MAX, AND, OR, ADD, etc.) on its two operands. The reductions are performed in parallel across all of the ALU's. The PRN interfaces with each processor through the IN and OUT registers. In order to satisfy certain correctness criteria established in [15], the basic unit of data transfer between the processors and the PRN is a *state vector.* A state vector consists of m elements, each of which has two components: a *value* and a *tag*. The idea is that the relevant

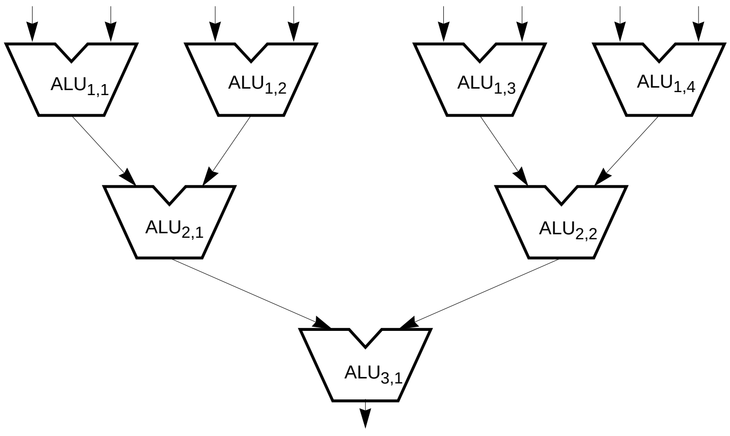

Figure 2 - A Parallel Reduction Network

synchronization information of a processor is encoded into these *m* elements so that global reductions may be performed on them. At each processor, the IN registers present a state vector to the PRN. The PRN starts the reduction by reading the first elements of each state vector and reducing them pair-wise at the top row of ALU's. Note the reduction operation is performed only on the *value* fields of the state vector elements. The *tag* simply accompanies the value. The top row of ALU's passes the reduced values down to the second row of ALU's. While the second row of ALU's reduces these values, the top row reduces the second elements of each state vector. Thus, the PRN operates in a pipelined fashion. The result of reducing all of the elements of all of the state vectors is a *globally reduced state vector* which is distributed back to the OUT registers of all of the processors.

The time required for a set of values to pass through a single PRN stage is called the *minor cycle time*, while the time required for the top row ALU's to read all the elements of the state vectors is called the *input cycle time*. Note an input cycle consists of *m* minor cycles, where *m* is the size of the state vectors. Because of the pipelined nature of the PRN, it takes $\log_2 n$ minor cycles for the global reduction of the first elements of a set of state vectors and thereafter, reductions of the remaining *m*-1 elements emerge from the PRN one minor cycle apart. Thus, a *reduction cycle*, which is the time required to produce a globally reduced state vector from a set of *n* input state vectors of size *m* each, is $(\log_2 n + (m-1))*c$, where *c* is the minor cycle time. Since speed of reduction is the primary design goal of this hardware, it is important that *c* be small. In the prototype hardware described in [15], *c* is 150 nanoseconds, giving a reduction cycle time of 1.2 microseconds for 32 processors with 4-element state vectors.

As mentioned earlier, tags simply accompany the values. At each ALU, the tags of the two operands are brought to a selector switch. The choice of which tag is propagated down to the next stage is made by a control signal from the ALU. For example, if the ALU performs a selective operation such as a *minimum* on the values, the tag of the smaller of the two values will proceed downward. In non-selective operations, the choice is arbitrary but deterministic.

The PRN has been designed such that it never blocks. It continuously reads state vectors from all of the IN registers, reduces them and distributes the output state vector back to the OUT registers. If a new state vector is not submitted by a processor during a reduction cycle, the same state vector is re-used in the next reduction cycle. Consequently, the output of the PRN is a stream of globally reduced state vectors, a new one being produced every *m* minor cycles.

To manage the high-frequency data coming out of the PRN, an *auxiliary processor* (AP) is assigned to each *host processor* (HP). The HP's perform all simulation specific tasks (executing events, sending and receiving messages, saving state, etc.) while the AP's perform all synchronization tasks (i.e., they are responsible for submitting state vectors to the PRN, reading the output of the PRN and executing synchronization algorithms). When an HP causes its state to change (by executing an event, receiving a message, etc.) it communicates this change to its AP. The AP incorporates such changes into state vectors which it submits to the PRN for reduction. Also, the AP reads the globally reduced output state vector of the PRN and makes a portion of this available to the HP.

The interface between the HP and the AP consists of two unidirectional channels (implemented as dual ported RAM). For information flowing from the HP to the AP, it is required that none of it be lost and that the AP process the information in the order sent out by the HP. For these two reasons, the IN Buffer is a FIFO. In the other direction, the only requirement is that the most recent version of the AP's output be available to the HP. Accordingly, the OUT Buffer consists of a single cell buffer, which is repeatedly overwritten by the AP each time it presents new data to the HP. The dual-ported RAM used to implement these buffers is mapped into the memory of the HP through one port and the AP through the other. Thus, access to the IN and OUT buffers is by simple memory read and memory write operations.

As mentioned, the PRN and the AP's operate asynchronously. Occasionally, AP's present state vectors (not necessarily all at the same time) to the PRN for reduction. The PRN operates in reduction cycles, reducing a set of state vectors, one from each processor, to a single output state vector in each cycle. Subsequent reduction cycles re-use state vectors for those AP's which do not present new state vectors. To satisfy these requirements, the design includes a custom interface between the AP's and the PRN. The AP's write new state vectors to the IN registers and read globally reduced state vectors from the OUT registers. Likewise, the PRN reads state vectors from the IN registers, reduces them and writes the global output to the OUT registers. The IN and OUT registers are comprised of three sets of registers each which provide the isolation between the AP and the PRN. The detailed operation of these register interfaces is described in [15].

For illustrations of the use of the hardware for computing synchronization values, the reader is referred to the original framework synchronization algorithms proposed in [14] and the double handshake acknowledgment protocol described in [10].

3 GVT algorithm

We assume familiarity with the Time Warp protocol and associated terminology. By definition, GVT is the minimum of two values: the smallest of the logical clocks of all of the LP's and the smallest unreceived message time among all of the LP's. Correspondingly, our GVT algorithm, listed in Figure 3, maintains the following two *T-values* at each LP$_i$:

σ_i: the logical clock of LP$_i$

υ_i: the minimum of the timestamps of all unreceived messages sent by LP$_i$

and the PRN is used to compute and disseminate:

$$\sigma' = MIN_{LP_i}(\sigma_i)$$

$$\upsilon' = MIN_{LP_i}(\upsilon_i)$$

```
HOST_PROC: IF     there are events in the events list
           THEN   local_clock = timestamp of next event
                  Enqueue (NEW_CLOCK, local_clock);
                  Execute event; Perform SEND_MSG if required;
                  Optionally save state;
           WHILE  there are newly received messages with timestamps < local_clock
                  Perform ROLLBACK;
           FOR    each new message
                  Perform RCV_MSG;
           Optionally collect fossils;

SEND_MSG:  Enqueue (SENT_MSG, message_time, message_id);
           Send the message;
           IF     message_sign > 0
                  Add antimessage to output list;

ROLLBACK:  local_clock := rollback time;
           Enqueue (NEW_CLOCK, local_clock);
           Roll back the execution of all events with time > rollback time;
           Restore state from the last time it was saved before rollback time;
           Rebuild state up to rollback time if required;
           FOR    each antimessage in output list with time > rollback time
                  Perform SEND_MSG;
                  Delete it from the output list;

RCV_MSG:   IF     message_sign > 0
           THEN   Insert corresponding event into events list;
           ELSE   Delete the positive event;
           Enqueue (RCVD_MSG, message_time, message_id);
```

Figure 3a - Host processor algorithm for GVT computation

so that GVT may be computed by each LP as $GVT = MIN(\sigma', \upsilon')$. The algorithm is shown in two parts: one to be executed at each HP and the other to be executed at each AP. For exposition, we assume that only one LP is assigned to each HP-AP pair. The HP algorithm performs the simulation specific tasks such as executing events, sending and receiving messages, saving state, etc. Thus, it maintains the logical clock of the LP, the events list, the list of antimessages (output list) and the state of the LP. The AP algorithm maintains the two T-values and manages the data flow into and out of the PRN. Note the σ_i maintained by an AP tracks the actual logical clock maintained by the corresponding HP. As the HP executes events, its state changes and those changes that affect GVT (such as changes in its logical clock value, sending a message, receiving a message, etc.) must be communicated to its AP so that they may be included in the GVT computation in the PRN. Recall that the communication channel between an HP and its AP is functionally a FIFO. To communicate a change to its AP, an HP enqueues a labelled communication in this FIFO. The label provides a means to identify the kind of state change being communicated. In the algorithm for the HP, this communication is indicated by the statement `Enqueue (label, value`$_1$`, value`$_2$`, ...)`.

The AP is responsible for three major tasks:

- it must read the output of the PRN, compute GVT as MIN(σ', υ') and write it to the OUT buffer (Figure 1)

- it must perform acknowledgments of messages through the PRN

```
AUX_PROC:  Read the PRN output;
           Write $MIN(\sigma', \upsilon')$ to the OUT buffer if changed;
           Perform DO_ACK;
           IF    FIFO is not empty
           THEN  Get next entry from FIFO;
                 CASE (entry):
                     (NEW_CLOCK, new_clock_value):
                                $\sigma_i$ := new_clock_value;

                     (SENT_MSG, message_time, message_id):
                                IF message_time < $\upsilon_i$
                                THEN $\upsilon_i$ := message_time;
                                Add message to outstanding message list;

                     (RCVD_MSG, message_time, message_id):
                                [IF $\rho_i = \{\infty, \Phi, 0\}$
                                THEN $\rho_i$ := (message_time, message_id, 1)]
                                ELSE Add message to unacknowledged message list;

DO_ACK:    IF    [ $\tau' = \rho_i$ AND $\rho_i \neq \{\infty, \Phi, 0\}$]
           THEN  Remove next batch to be acknowledged
                     from the unacknowledged message list;
                 Set $\rho_i$ to acknowledge this batch;

           [IF   $\rho'$ has been sent to me
           THEN  IF  messages in $\rho'$ batch are in outstanding message list
                 THEN $\tau_i$ := $\rho'$ ];
                     Mark them as acknowledged;
                     Remove any other batches marked as acknowledged
                                from outstanding message list;
                     IF    timestamp of acknowledged batch = $\upsilon_i$
                     THEN  $\upsilon_i$ := smallest timestamp in outstanding message list;
                 ELSE $\tau_i$ := $\{\infty, \Phi, 0\}$;
           ELSE  $\tau_i$ := $\{\infty, \Phi, 0\}$;
```

Figure 3b - Auxiliary processor algorithm for GVT computation

- it must process as soon as possible the labelled communications that its HP inserts into the FIFO (IN buffer in Figure 1)

The second task warrants some explanation. Since messages are timestamped, messages in transit (those that have not yet been received) represent timestamps in transit (those that may not be reflected in the logical clock of any LP). To account for these timestamps, the algorithm maintains υ_i which is the minimum of the timestamps of all messages sent by LP$_i$ which have not yet been received and acknowledged. The simplest way to maintain υ_i is to have the LP's acknowledge every message when it is received. One of the novelties of our GVT computation algorithm is that messages are acknowledged through the PRN and not through the host communication network. Thus, no additional traffic is introduced into the simulation message traffic. Clearly, it is the AP's which must perform the acknowledgments through the PRN. Message acknowledgments are performed using a two-phase protocol described later. This protocol requires the use of two more T-values: ρ_i and τ_i. Their use will also be explained later. Thus, each AP has a four-element state vector consisting of the four T-values: σ_i, υ_i, ρ_i and τ_i.

It is important to note here that in order to maintain υ_i correctly, the timestamps of antimessages must also be part of the GVT computation. To see why, consider a system of two LP's with LP$_0$ having σ_0 at 1000, LP$_1$ having σ_1 at 1500 and no messages in transit. Therefore, GVT has a value of 1000. Now, LP$_0$

sends LP_1 an antimessage with timestamp 1000 and then proceeds to execute its next event with timestamp 2000. While the antimessage is in transit, if antimessages are not acknowledged, we have σ_0 at 2000, σ_1 at 1500 and both υ_0 and υ_1 at ∞, giving a GVT of 1500. When the antimessage is finally received by LP_1, σ_1 falls to 1000, bringing GVT down to 1000. This is an error since GVT must be strictly non-decreasing. The error occurs because the smallest timestamp in the system, which is the timestamp of the antimessage, is momentarily absent from the GVT computation process. The problem is solved by requiring antimessages also to be acknowledged.

3.1 Host processor algorithm

The algorithm to be executed by each HP is shown in Figure 3a. Recall that the HP performs all of the actions associated with simulating the system and with Time Warp. Whenever its actions affect GVT, it communicates with its AP by enqueueing a labelled communication in its FIFO. The algorithm consists of a main procedure called `HOST_PROC` and three subroutines: `SEND_MSG`, `ROLLBACK` and `RCV_MSG`. In this algorithm, we adopt aggressive cancellation [13] and also assume that messages do not preempt the execution of events. However, we have indicated elsewhere [18] how to accommodate lazy cancellation and event preemption by messages into this GVT computation algorithm.

HOST_PROC

The main program is simply an event execution loop. As a result of executing an event, an LP may send out messages. Sending of messages is done using the `SEND_MSG` procedure. At the end of an event execution, the HP determines if any messages have been received in its logical past. If so, it initiates a rollback by calling the `ROLLBACK` routine. Since more rollback-causing messages may be received while the HP is rolling back, the `ROLLBACK` routine is called repeatedly until all received messages are in the HP's logical future (i.e. it has rolled back far enough). These messages are then incorporated into the events list by calling `RCV_MSG` for each received message. State is optionally saved after event execution and fossils are collected optionally before event execution. As a result of executing events, the HP's logical clock advances. Each such advance is accompanied by an `Enqueue` operation to inform the AP of the new clock value. All other state changes which affect GVT (and therefore the corresponding communications) occur in the other three routines. Whenever the HP requires the value of GVT (e.g. for committing events, collecting fossils, etc.) it reads the value of GVT written by the AP in the OUT buffer (Figure 1).

SEND_MSG

Since the sending of a message may affect LP_i's υ_i, the HP indicates a sent message to its AP by enqueueing a communication prior to sending the message. This communication includes the timestamp of the message and a message identifier so that the AP can identify the message. After the message is sent, if the message is a positive (true) message, an antimessage is constructed for the same and appended to the output list.

ROLLBACK

This is a typical Time Warp rollback procedure with two additions. First, since a rollback changes the logical clock value of the HP, the new clock value is communicated to the AP by an `Enqueue` operation. Second, since antimessages must also be acknowledged (as explained earlier), antimessages are also sent using the `SEND_MSG` routine. This ensures that the AP is notified of the sending of an antimessage and therefore expects an acknowledgment for the same.

RCV_MSG

Recall that this routine is called only after the HP has rolled back far enough so that all received messages are in the HP's logical future. Thus, depending on whether the message is a positive message or an antimessage, a new event is created or an existing one is deleted. Finally, the HP informs its AP that it has received a message and also sends the AP the timestamp of the message and its identifier.

3.2 Auxiliary processor algorithm

In Section 3, we listed three tasks that the AP must perform. It is important that all of these are performed often and none is delayed arbitrarily. For this reason, the algorithm for the AP (Figure 3b) consists of a single loop in which each of the three tasks is performed. The algorithm is separated into a main routine, `AUX_PROC` and a subroutine, `DO_ACK`. The main routine calls `DO_ACK` to perform the acknowledgment of messages through the PRN. Segments of the algorithm enclosed within [...] are to be executed atomically. We describe the `DO_ACK` routine first and then the main routine `AUX_PROC`.

DO_ACK

Each AP$_i$ maintains two lists of messages: the *outstanding message list* which includes those messages sent by LP$_i$ for which acknowledgments have not yet been received and the *unacknowledged message list* which includes those messages received by LP$_i$ which it has not yet acknowledged. When a message is sent, it is outstanding until an acknowledgment is received for it and is therefore inserted into the outstanding message list. When an acknowledgment is received for it, it is deleted from the outstanding message list. Since all acknowledgments are performed through the same PRN, any AP can perform at most one acknowledgment at a time. While this acknowledgment is in progress, newly received messages are inserted into the unacknowledged message list to be acknowledged later. Before a message is acknowledged, it is deleted from the unacknowledged message list. Entire messages are not stored in the lists; instead, for each message, its timestamp and message identifier are stored. The acknowledgment scheme requires that each message be associated with a globally unique identifier and that these message identifiers form a contiguous sequence of numbers for each ordered pair of communicating LP's. One way of assigning such message identifiers in a distributed manner with no overhead is by having the message identifier be the concatenation of the identifiers of the sending and receiving LP's and a sequence counter value. An LP maintains a separate sequence counter for each LP that it sends messages to. Prior to sending a message to LP$_j$, LP$_i$ increments the counter associated with LP$_j$ and uses this value in the identifier for the message. The outstanding message list of LP$_i$ is actually implemented as several lists, one for each LP that LP$_i$ sends messages to. Similarly, the unacknowledged message list is implemented as several lists, one for each LP that can send messages to LP$_i$. All of these lists are sorted in ascending order of the sequence counter value field of the message identifier. This organization makes it possible to perform batched acknowledgments as discussed below.

We now describe the two-phase acknowledgment protocol presented in [10]. One of the novelties of this protocol is that it acknowledges multiple messages with a single physical acknowledgment. This is referred to as *batched acknowledgments* - each physical acknowledgment corresponds to a batch of messages. Each batch is represented by the message in the batch with the smallest timestamp, its identifier and the size of the batch. The protocol uses two T-values ρ_i and τ_i. Each of these consists of three fields: the timestamp of the representative message of the batch, its identifier and the number of messages in the batch. The global counterparts of these T-values are defined as:

$$\rho' = \langle \rho_j | (timestamp(\rho_j) = MIN_{LP_i}(timestamp(\rho_i))) \rangle$$

$$\tau' = \langle \tau_j | (timestamp(\tau_j) = MIN_{LP_i}(timestamp(\tau_i))) \rangle$$

To acknowledge a batch of messages sent to it by LP$_s$, LP$_r$ initiates the first phase by setting its ρ_r to the timestamp and message identifier of the batch. Without loss of generality, assume that the timestamp of ρ_r is the smallest among all other acknowledgments submitted simultaneously (i.e, all ρ_i). Thus, ρ' will equal ρ_r. When LP$_s$ observes this (it is guaranteed to do so ultimately since LP$_r$ does not change ρ_r), it knows that a batch of its messages to LP$_r$ is being acknowledged. It deletes this batch from its outstanding message list and the first phase is complete. LP$_s$ initiates the second phase by setting its τ_s to ρ'. After some time, τ' will equal τ_s and therefore ρ_r. When LP$_r$ observes this (it will do so ultimately because LP$_s$ does not change τ_s), it knows that LP$_s$ has seen its acknowledgment. LP$_r$ then removes its acknowledgment by changing ρ_r. After some delay, this change is reflected in the output of the reduction network as a change in ρ'. When LP$_s$ observes this change, it knows that LP$_r$ has removed its acknowledgment and so it *relinquishes* the secondary acknowledgment by setting τ_s to $\{\infty, \Phi, 0\}$ and the second phase is complete. We note here that the two phases are required because AP's may not observe all state vectors emerging from the PRN. At any time during these two phases, a new acknowledgment with a timestamp smaller than ρ_r may be submitted. In such a case, the protocol preempts the current acknowledgment, completes the new one and then restarts the preempted acknowledgment. We have shown in [18] that this protocol correctly acknowledges all messages in a finite amount of time

AUX_PROC

The AP begins by reading the output state vector of the PRN. If MIN(σ', υ') has changed from the last iteration, the AP writes the new value to the OUT buffer of the HP-AP interface. It then calls `DO_ACK` to check for acknowledgments in the newly obtained output state vector. Finally, it processes a labelled communication from its HP. As seen in Section 3.1, an HP communicates with its AP in three situations: when its logical clock value changes, when it sends a message and when it receives a message. These communications are correspondingly labelled `NEW_CLOCK`, `SENT_MSG` and `RCVD_MSG`.

`NEW_CLOCK`: The AP sets its σ_i to the new value of the logical clock supplied in the communication.

`SENT_MSG`: Since the message sent by the HP is outstanding initially, the AP sets its υ_i to the minimum of its old value and the timestamp of the message. Also, it adds the message to its outstanding message list.

`RCVD_MSG`: If the AP is not acknowledging a message currently, it starts acknowledging the message in the labelled communication. Otherwise, it inserts the message into the unacknowledged message list for acknowledging later.

3.3 Correctness

The two problems faced by any GVT computation scheme [7] are solved in our method as follows: (i) the simultaneous reporting problem is solved by the hardware design which takes an atomic snapshot of the global state as presented by the AP's using the state vectors (ii) the transient message problem is solved by acknowledging every message through the PRN. The asynchronous nature of the operation of the HP, AP and the PRN and the latencies due to the interfaces between these components introduces the possibility of race conditions in the GVT algorithm. A detailed proof of correctness of this algorithm which is too elaborate to include in this paper is presented in [18]. The basic intuition behind this proof is that at all times, the algorithm maintains the following invariant: the smallest timestamp in the simulation (actual GVT) is represented in at least one LP's σ_i or υ_i and therefore in the computed GVT.

3.4 Initialization

At system start-up, the AP's are brought up in a "listening" mode. The program for the AP's is downloaded from the HP's. One HP-AP pair is designated the *master* pair and is responsible for initializing the controllers for the PRN. This is also done by downloading initialization information from the master HP to the master AP and then to the PRN. Once initialized, the PRN starts reducing state vectors. At each LP$_i$, the four elements of the state vector are initialized as follows: σ_i is set to 0, υ_i is set to ∞ and ρ_i and τ_i are both set to $\{\infty, \Phi, 0\}$ where Φ represents the null value for the identifier. The execution of the GVT algorithm commences after enough time has elapsed to ensure that the output of the PRN contains the results of reductions on the initialized state vectors.

4 Performance

We present the results of preliminary studies of the performance of our GVT algorithm.

4.1 Speed

In [17], we have described an experiment which simulated eight LP's performing a busy-work application using Time Warp. The LP's used the GVT algorithm presented here to compute GVT. The framework hardware and the GVT algorithm were simulated in significant detail. The hardware was simulated at the component level (AP, ALU, buffers, etc.) and the software was modeled as the execution of the GVT algorithm on the AP's and the HP's. For this model, estimates of timing parameters such as minor cycle time, memory read time, etc. were obtained from the prototype being built at the University of Virginia.

The model was designed to compute the average amount of real time for which computed GVT lags behind actual GVT. Basically, the model measured the average time it takes for a change made by an HP which affects the actual value of GVT to appear in the computed value of GVT. From the results of several simulation runs, we concluded that the average lag was around ten microseconds (approximately three orders of magnitude smaller than typical event execution times). Although rough estimates were used for timing parameters which were difficult to determine, we believe that in actuality, this lag will remain in the few tens of microseconds. The prototype hardware is almost functional. We expect to implement this GVT algorithm and obtain actual performance measurements by the Summer of 1993.

4.2 Scalability

The main benefit of using a tree structure for the PRN is that it scales well with the number of processors. As the number of processors, n, is increased, the time to compute a globally reduced state vector increases as $\log_2 n$. For instance, if the number of processors is doubled from 32 to 64, the time to compute a four-element global state vector increases from 1.2 microseconds to only 1.35 microseconds. However, with an increased number of processors, the volume of traffic to the PRN will also increase and it is reasonable to expect the increase in the time to acknowledge messages to be more than logarithmic. In spite of this, we believe that the numbers for larger systems (few hundreds of processors) will be on the same order of magnitude as those predicted by the simulations.

5 Conclusion

We have presented a new method of computing GVT using inexpensive hardware designed to support synchronization in parallel computations. All of the GVT computation algorithms proposed thus far interfere with the simulation and tend to increase the completion time of the simulation. Our method is unique in that the GVT computation does not interfere in any way with the simulation. The only responsibility of the LP's is to communicate some state changes to the hardware, which involves writing a few words to memory. GVT is made available to each LP asynchronously of the other LP's. Each LP obtains the value of GVT by reading a memory location. Thus GVT is available to the LP's at no cost. Simulations have predicted that the GVT provided thus will be very accurate. Finally, the hardware and therefore the algorithm will scale very well with the number of processors in the system.

In summary, the main reasons why we believe our algorithm will outperform previous algorithms are:

- high-speed, scalable PRN used to compute GVT asynchronously

- dedicated processors manage GVT computation
- messages acknowledged in the PRN

Since a prototype of the hardware is nearly functional, we expect to have actual performance measurements in the near future.

Acknowledgments

This work was supported in part by the National Science Foundation (grant CCR-9108448, Aug. 91, number 48), MITRE Corporation (Academic Affiliates Program) and Mystech, Inc. (Academic Affiliates Program).

REFERENCES

1. Bauer, H., and Sporrer, C., Distributed logic simulation and an approach to asynchronous GVT-calculation. *Proceedings of the sixth workshop on parallel and distributed simulation* (January 1992), 205-208.
2. Bellenot, S., Global virtual time algorithms. *Proceedings of the 1990 SCS Multiconference on Distributed Simulation* (January 1990), 122-127.
3. Concepcion, A.I., and Kelly, S.G., Computing global virtual time using the Multiple-Level Token Passing algorithm. *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation* (January 1991), 63-68.
4. Filloque, J.M., Gautrin, E., and Pottier, B., Efficient global computations on a processor network with programmable logic. Research Report number 1374, Institut National de Recherche en Informatique et en Automatique (January 1991).
5. Fujimoto, R.M., “Parallel discrete event simulation”, *Communications of the ACM*, Vol. 33, No. 10 (October 1990), 30-53.
6. Jefferson, D.R., “Virtual time”, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, (July 1985), 404-425.
7. Lin, Y-B., and Lazowska, E.D., Determining the global virtual time in a distributed simulation. Technical Report number 90-01-02, Department of Computer Science and Engineering, University of Washington (December 1989).
8. Livny, M., and Manber, U., Distributed computation via active messages, *IEEE Transactions on Computers* Vol. C-34, No. 12 (December 1985), 1185-1190.
9. Mattern, F., Efficient algorithms for distributed snapshots and global virtual time approximation. To appear in the special issue of *Journal of Parallel and Distributed Computing* on parallel and distributed simulation, August 1993.
10. Pancerella, C.M., “Improving the efficiency of a framework for parallel simulations”, *Proceedings of the 6th Workshop on Parallel and Distributed Simulation* (January 1992), 22-27.
11. Preiss, B., The Yaddes distributed discrete event simulation specification language and execution environments. *Proceedings of the SCS Multiconference on Distributed Simulation* (January 1989), 139-144.
12. Preiss, B.R., MacIntyre, I.D., and Loucks, W.M., On the trade-off between time and space in optimistic parallel discrete-event simulation. *Proceedings of 6th Workshop on Parallel and Distributed Simulation* (January 1992), 33-42.
13. Reiher, P., Fujimoto, R., Bellenot, S., and Jefferson, D., Cancellation strategies in optimistic execution systems. *Proceedings of the 1990 SCS Multiconference on Distributed Simulation* (January 1990), 112-121.
14. Reynolds, P.F., Jr., An efficient framework for parallel simulations. *International Journal in Computer Simulation*, Vol. 3, No. 4, 1992.
15. Reynolds, P.F. Jr., Pancerella, C.M. and Srinivasan, S., Design and performance analysis of hardware support for parallel simulations. Technical Report number TR-92-20, Computer Science Department, University of Virginia, June 1992. To appear in the special issue of the *Journal of Parallel and Distributed Computing* on Parallel and Distributed Simulation, August 1993.
16. Samadi, B., Distributed simulation: algorithms and performance analysis. In *Distributed Processing* (IFIP), ed. M. H. Barton, E.L. Dagless and G. L. Reijns, Elsevier Science Publishers (North-Holland), 19-34.
17. Srinivasan, S., Modeling a framework for parallel simulations. Master of Science Thesis, Computer Science Department, University of Virginia (May 1992).
18. Srinivasan, S., and Reynolds, P.F. Jr., Hardware support for aggressive parallel discrete event simulations. Technical Report number TR-93-07, Computer Science Department, University of Virginia, February 1992.
19. Tomlinson, A.I., and Garg V.K., An algorithm for minimally latent global virtual time. *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation* (May 1993), 35-42.