*Proceedings of the 1999 Winter Simulation Conference*
*P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds.*

# OPTIMISTIC PARALLEL SIMULATION OVER A NETWORK OF WORKSTATIONS

Reuben Pasquini
Vernon Rego

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, U.S.A.

## ABSTRACT

The low cost and scalability of a PC and ethernet-based network of workstations (NOW) makes the NOW an attractive platform for parallel discrete event simulation (PDES). This paper discusses the demands a parallel simulation places upon a network that connects distributed workstations, and presents two approaches to managing inter-processor communication in PDES on a NOW.

## 1 INTRODUCTION

A discrete event simulation (DES) uses a computer to test a model of a system whose state changes at discrete points in time. A simulation operates on a model's *state* variables during each of a sequence of time-ordered *events*. A parallel discrete event simulation (PDES) attempts to speed up the execution of a DES by distributing the simulation's workload between multiple processors. Parallel simulation holds great promise for meeting the simulation needs of developers of increasingly complex systems.

A network of workstations is an inexpensive and widely available platform for PDES. A NOW usually consists of several workstations or PC's connected by an ethernet. A NOW has advantages and disadvantages when compared to a multiprocessor (MP) like the IBM SP2 or SGI Origin. A NOW, based on commodity hardware and software, is inexpensive and easy to upgrade. Today, each node of a NOW (workstation) is just as computationally powerful as a node of an MP since most MP systems use the same processor found in workstations. However, the interconnection network in a multiprocessor supports communication with higher bandwidth, lower latency, and stronger reliability guarantees than the typical ethernet that interconnects the nodes in a NOW.

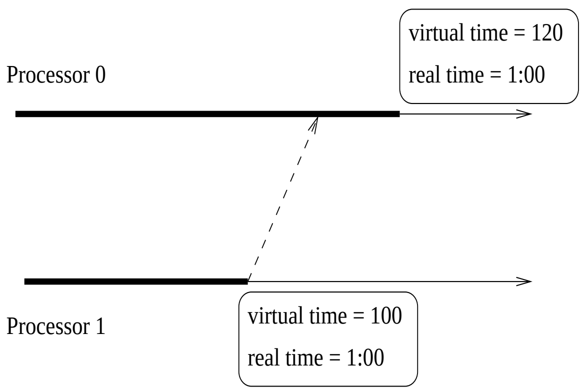
Experiments with the ParaSol PDES system indicate that a parallel simulation with tight interprocessor coupling must regulate its rate of interprocessor communication to run well on a NOW. ParaSol is an experimental process- and object-oriented parallel simulation library for distributed-memory multiprocessors and workstation clusters. This paper explores techniques for regulating interprocessor communication (IPC) between processors participating in a parallel simulation on a network of workstations.

### 1.1 PDES Concepts

A discrete event simulation executes a time-ordered sequence of simulation *events*. Each event object has a *time-stamp* and a *handler*. The simulation uses an event's time-stamp to schedule the event's execution. A simulation executes events in nondecreasing time–stamp order so that *virtual time* (the time–stamp on the last executed event) never decreases. The simulation calls an event's *handler* method to execute the event. During its execution, an event may access simulation objects and schedule future events.

Parallel simulation attempts to speedup a simulation's execution by distributing the simulation's events and objects across multiple processors. Given $N$ processors and $M$ events, each processor would ideally handle $M/N$ events, suggesting an ideal speedup of $N$. Unfortunately, distributed events may not access simulation objects in time–stamp order. For example, processor $P_1$ may execute an event $E_a$ with time–stamp $T_{E_a} = 16$ after processor $P_2$ executes an event $E_b$ with time–stamp $T_{E_b} = 36$. If $E_a$ generates an event $E_c$ with time–stamp $T_{E_c} = 21$ that $P_2$ must execute (since $E_c$ accesses a simulation object $O_{P_2}$ located on processor $P_2$), then $E_c$ accesses $O_{P_2}$ after $E_b$ even though $T_{E_c} < T_{E_b}$.

A PDES must execute events in a *causally consistent* way. A simulation is causally consistent if events access each simulation object in nondecreasing time–stamp order. The time warp algorithm (Jefferson 1985) is an example of an *optimistic* algorithm for PDES. Time warp is optimistic in the sense that each processor $P_0$ executes events in time–stamp order under the optimistic assumption that causality is not being violated. At any point, however, $P_0$ may receive a *straggler* event $E_s$ (from another processor) that should have been executed before the last several events already executed by $P_0$ (see Figure 1). When $P_0$ receives the straggler $E_s$, $P_0$ *rolls back* to a check-pointed system state that corresponds to a time–stamp which is less than the straggler's time–stamp. Processor $P_0$ resumes its execution from this point, and $P_0$ processes the straggler $E_s$ in the right time–stamp order. A successful optimistic PDES minimizes the runtime costs of *state-saving* system state (for potential rollback), *rollback* (to recover state when a straggler arrives), *global virtual time* (gvt) computation (to determine a global minimum on the simulation's virtual time), and *interprocessor communication* (IPC).




Processor $P_1$ generates an event for processor $P_0$ at virtual time 100, but $P_0$ has already reached v.t. 120. Processor $P_0$ will roll back when $P_0$ receives the straggler event.

Figure 1: Causality Error in an Optimistic Parallel Simulation

Several parallel simulation systems are in use today in experimental and applied settings. The *GTW* system is an optimistic event-based system developed at Georgia Tech (Penesar and Fujimoto 1997). *ParSec* is a conservative system developed at UCLA (Bagrodia et al. 1998). *Warped* is an optimistic system developed at the University of Cincinnati (Chetlur et al. 1997). *APOSTLE* is a process-based simulator that uses the breathing time-buckets algorithm (Booth and Bruce 1997). The results in this paper are based on experiments carried out with PARASOL, an optimistic simulator under development at Purdue University (Mascarenhas, Knop, and Rego 1997). For the remainder of this paper the terms "parallel simulation" and "PDES" both refer to optimistic parallel discrete event simulation.

### 1.2 Communication and PDES

Fast, timely communication is necessary to achieve good performance in parallel simulation. Distributed processors executing a PDES exchange messages in at least three situations. First, when a processor $P_0$ schedules an event $E_a$ on a remote processor $P_1$, $P_0$ sends $P_1$ a message. If $P_0$ later receives a straggler event from a remote processor, then $P_0$ sends an anti-message to $P_1$ to cancel event $E_a$. Finally, each processor periodically exchanges virtual time information with every other processor to compute a new global virtual time (gvt). The gvt is a lower bound on the PDES virtual time.

Many PDES systems avoid problems with interprocessor communication by running on shared memory platforms. A shared memory architecture has important advantages over a distributed memory architecture for PDES. Shared memory allows processors to exchange data by passing pointers between each other rather than packing the data into a message to be sent over a network. A shared memory space also allows a user to view his model as a single unit rather than a collection of subunits that communicate with each other.

Distributed memory platforms have two advantages over shared memory platforms for PDES. First, a distributed memory system can exploit idle processors already available on a network of workstations to speedup a simulation. Second, a distributed memory multiprocessor can scale cheaply to a large number of processors to support parallel simulations with sufficient parallelism.

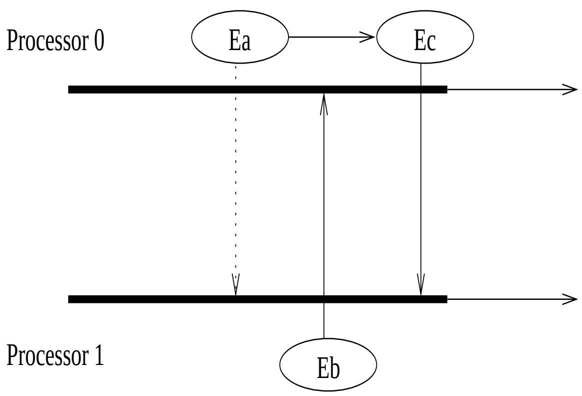
Distributed memory PDES systems rely upon a message passing library to provide high performance and platform independent management of interprocessor communication, synchronization, flow control, and buffer management. The *PVM* (Suderam et al. 1994) and *MPI* (1995) message passing libraries are two popular communication systems. Most message passing middleware is designed to address the needs of structured parallel applications which synchronize via blocking `send` and `receive` operations, but message passing systems which use multithreading to efficiently overlap communication and computation in asynchronous and soft realtime applications have recently become available (Gomez, Rego, and Sunderam 1997).

Optimistic parallel simulations exhibit unpredictable asynchronous communication patterns not well suited to the traditional synchronous `send` and `receive` communication paradigm. In PARASOL, a processor $P_0$ may send a message to a destination processor $P_1$ at unexpected times. Rather than synchronize $P_0$ and $P_1$ with blocking `send` and `receive`, PARASOL requires $P_0$ to send its message $M$ with a *nonblocking* `i_send`. Processor $P_1$ eventually receives $M$ since $P_1$ periodically polls the network for arriving messages. This scheme has two drawbacks. First, since the simulation cannot anticipate when a new message will arrive, the simulation must regularly poll the network for arriving messages within the simulation driver's event execution loop. Second, non-blocking `i_send` bypasses the communication system's flow control mechanisms. Therefore, a message sender can generate messages faster than the message passing system can deliver messages to receivers. In this way pending messages sent asynchronously (non–

blocking) can accumulate in the sender's memory space and eventually overwhelm the simulation.

Several approaches to improving interprocessor communication performance in PDES have been proposed by researchers. Chetlur et al. (1997) explore the benefits of batching messages in the *Warped* PDES system. This work explores the trade-off in message batching between the benefit of decreasing per-message communication overhead and the cost of increased message delivery latency. This trade-off is complicated in PDES by the potential for destructive interdependencies between messages in a batch. For example, suppose that processor $P_0$ has local virtual time $T_{P_0} = 9$ when $P_0$ generates a message $M_{E_a}$ that schedules an event $E_a$ with time-stamp $T_{E_a} = 10$ on processor $P_1$. Processor $P_0$ does not send $M_{E_a}$ immediately, since $P_0$ wants to batch $M_{E_a}$ with another message. Processor $P_0$ goes on to execute an event $E_b$ scheduled by a message $M_{E_b}$ sent from $P_1$ at virtual time $T_{P_1} = 11$. Processor $P_0$ should not execute $E_b$ since $P_1$ should have executed $E_a$ (the event that $M_{E_a}$ will schedule) before sending $M_{E_b}$. However, if $P_0$ does not notice this conflict, then $P_0$ may go on to generate a message $M_{E_c}$. Finally, $P_0$ batches $M_{E_c}$ with $M_{E_a}$ without realizing that a destructive dependency exists between the two messages in the batch (see Figure 2).

Penesar and Fujimoto (1997) describe an adaptive flow control mechanism for regulating the rate at which each processor generates events for other processors. Their adaptive algorithm computes a virtual time window that limits each processor's optimism so that no processor advances too far beyond the system gvt. In this way, the adaptive algorithm attempts to prevent a processor from generating an event that will later be canceled by an anti-message. Similarly, Ferscha (1995), Mascarenhas (1997), and others have explored adaptive synchronization algorithms that regulate




Processor $P_0$ batches messages for events $E_a$ and $E_c$ together even though $E_c$ depends on an event $E_b$ from $P_1$, and $E_b$ will be rolled back when $E_a$ arrives at $P_1$.

Figure 2: Message Batching in PDES

"optimism" in PDES. Adaptive synchronization allows each processor in a PDES to decide whether to execute its next event or wait to receive a message. A processor bases its decision on probabilistic assumptions about the rate of interprocessor communication.

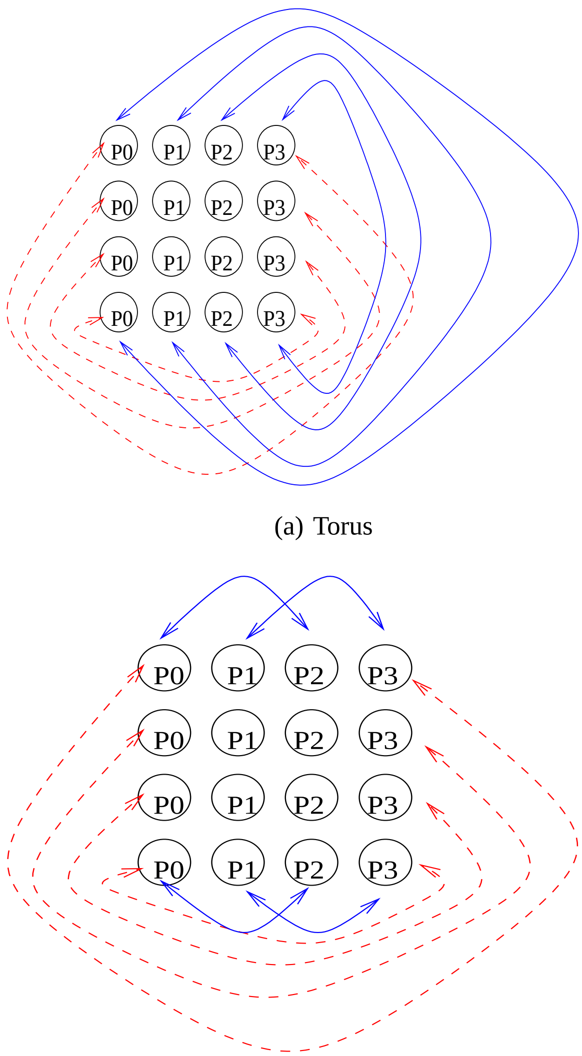
Finally, Damani, Wang, and Garg (1997) describe an algorithm that avoids cascading rollbacks by requiring each processor $P$ to stamp each message $M$ that $P$ sends with two Lamport clocks (Lamport 1978). If $P$ rolls back, then $P$ broadcasts a rollback-message with which the other processors can determine which of their received messages are valid and which are invalid. A shortcoming of this work is that it assumes the availability of an efficient and reliable broadcast mechanism even though most local area networks do not directly provide such support.

## 2 COMMUNICATION PROTOCOLS FOR PDES

The *torus* is an often used benchmark for measuring PARASOL's performance. The torus model consists of $N \times N$ servers arranged in a mesh that wraps around at its ends to form a doughnut. The experiment evenly distributes $N^2/2$ simulation processes over the mesh and then allows the processes (customers) to move randomly between neighboring servers $N^2$ times. A customer that arrives at a server requests to be serviced for an exponentially distributed service time. If the server is busy, then the server places the customer in a FIFO queue.

Experiments testing PARASOL's performance simulating other models revealed that PARASOL could not even complete a simulation of a baseball queueing model on a network of workstations (NOW). Like the torus, the baseball model consists of an $N \times N$ mesh of servers. Unlike the torus, the baseball connects the ends of the mesh to form a ball (rather than a doughnut), and the baseball allocates $N^2$ customers (rather than $N^2/2$ customers). The ball shape means that each processor simulating a baseball communicates with up to three neighbors (rather than two neighbors), and doubling the number of customers doubles the amount of interprocessor communication. Figure 3 shows diagrams of $4 \times 4$ torus and baseball models whose objects are evenly distributed between four processors ($P_0$, $P_1$, $P_2$, $P_3$).

An investigation into the reason for PARASOL's difficulty simulating the baseball model reveals that the simulation generates messages faster than the network can deliver messages. Since a parallel simulation generates messages at random points in time, a message sender may not synchronize with a message receiver without risking deadlock. For example, suppose processor $P_0$ sends a message $M_0$ to processor $P_1$ with MPI's normal blocking `send` routine. Processor $P_0$ may block on the `send` until $P_1$ receives $M_0$, depending on MPI's flow-control algorithm. Ideally, $P_1$ eventually uses MPI's non-blocking `i_receive` or `probe` routines to receive $M_0$. However, if $P_1$ sends a




(a) Torus

(b) Baseball

Figure 3: Torus and Baseball Models

message $M_1$ to $P_0$ before receiving $M_0$, then $P_1$ may block on `send`. The PDES is deadlocked in this situation since $P_0$ and $P_1$ are both blocked in `send` operations. To avoid this kind of deadlock, PARASOL allows a processor to communicate with another processor only with non-blocking `i_send` and `i_receive` operations. PARASOL bypasses the communication system's flow control mechanisms when PARASOL uses non-blocking `i_send`. Therefore, a sender can generate messages faster than the message passing system can deliver the messages to receivers. These messages accumulate in the sender's memory space, and eventually overwhelm the simulation.

This paper compares two approaches to communication in PDES that impose flow control on the simulation's message traffic. The *flow-controlled time-warp protocol* (FTWP) does not allow message senders to generate messages faster than receivers process messages. When a processor $P_0$ sends a message $M$ with MPI's non-blocking `i_send` routine, `i_send` returns a handle $H_M$ that $P_0$ can test to determine when MPI has safely delivered $M$ to $M$'s destination. Processor $P_0$ places each handle $H_M$ in a send-list. Processor $P_0$ periodically tests each handle in the send-list, and discards every handle $H_M$ whose message $M$ has been delivered. The FTWP simply requires a processor $P_0$ to stop simulating new events when $P_0$'s send-list grows beyond a fixed size (five handles in PARASOL's implementation of FTWP). Processor $P_0$ can resume simulating events as soon as the network delivers enough of the outstanding messages in the send-list.

The *warp-token protocol* (WTP) imposes order on PDES interprocessor communication on a NOW by requiring the processors to take turns sending messages. The WTP circulates a token between the processors participating in a parallel simulation. A processor $P_0$ can send a message only when $P_0$ holds the *token*. Therefore, each message $M$ that $P_0$ generates is stored in a send-queue until $P_0$ receives the token. When $P_0$ receives the token, $P_0$ bundles every message in its send-queue into the payload of a *token-message* $K$. Processor $P_0$ then broadcasts $K$ to the other processors. The number of messages in $K$'s payload (the batch size) is therefore a function of the rate at which $P$ generates messages and the token circulation time. When processor $P_r$ receives $K$, $P_r$ unpacks each component messages $M$ in $K$'s payload. If $P_r$ is the destination for $M$, then $P_r$ executes $M$'s handler routine. Only the token holder can send a token message, and every token message is broadcast to every processor. The token holder can send only one token message per possession, and the token moves between processors in a predefined order.

## 2.1 GVT

A parallel simulation's global virtual time (gvt) is the minimum of the local virtual time (lvt) on each processor and the time-stamp on every message in transit between processors. An optimistic parallel simulation must periodically compute gvt so that each participating processor can reclaim the memory allocated to checkpoint buffers. Since a processor cannot rollback to a virtual time preceding gvt, each checkpoint buffer saving state with virtual time smaller than gvt can be safely reclaimed by a processor. The process of reclaiming old checkpoint buffers is called *fossil collection*.

Most gvt algorithms require each processor to report its lvt to a leader who computes the new gvt and broadcasts the result. A PDES that employs such an algorithm must balance the communication cost of gvt calculation with the memory cost of delayed fossil collection to select a frequency for gvt calculation.

The WTP has the benefit of making gvt computation simple, frequent, and inexpensive. The warp token protocol's gvt algorithm Requires each processor $P_0$ to maintain a Lamport clock $G_{P_0}$ that tracks the lvt on each processor in the simulation. A Lamport clock is simply an array with an entry for each processor. When $P_0$ receives a token message $K$, $P_0$ looks at the time-stamp on $K$ to determine the lvt $T_s$ at the processor $P_s$ that sent $K$, and $P_0$ sets $G_{P_0}[s] = T_s$. Next, $P_0$ looks at the time-stamp $T_r$ on each message $M_r$ from $P_s$ to processor $P_r$ packed in $K$'s payload. If $T_r < G_{P_0}[r]$, then $P_0$ sets $G_{P_0}[r] = T_r$. After processing every message $M_r$ in $K$'s payload, $P_0$ knows the gvt is $gvt = min(G_{P_0}[r])$, the smallest virtual time in $G_{P_0}$.

### 2.2 Message Cancellation

When a processor $P_0$ in a PDES rolls back, $P_0$ sends anti-messages to cancel messages that $P_0$ sent during the period being rolled back. If an anti-message $A_i$ sent to processor $P_1$ by $P_0$ to cancel message $M_i$ does not arrive until after $P_1$ has processed events triggered by $M_i$, then $P_1$ is forced to rollback its computation. When $P_1$ rolls back, $P_1$ may be forced to send its own anti-messages which may in turn cause more rollbacks. This phenomenon, called time-warp thrashing or cascading rollbacks, can significantly slow the parallel simulation.

The WTP avoids time-warp thrashing by eliminating the need for anti-messages. Each processor $P_0$ keeps a Lamport clock $C_{P_0}$ to track $P_0$'s dependencies on other processors. For example, if $P_0$ receives a message from $P_1$ that schedules an event $E_{33}$ at virtual time 33, then $P_0$ updates $C_{P_0}$ so that $C_{P_0}[1] = 33$ just before executing $E_{33}$. When $P_0$ generates a message $M$, $P_0$ attaches a copy of $C_{P_0}$ to $M$ before placing $M$ in the send-queue (to later be bundled into a token $K$). If $P_0$ rolls back, $P_0$ must roll $C_{P_0}$'s state back. Therefore, $C_{P_0}$ is a state-saved object.

When $P_0$ receives a token message $K$, $P_0$ handles each message $M_0$ in $K$'s payload whose destination is $P_0$, and $P_0$ places $M_0$ onto a list for received messages. This message is fossil collected when the gvt advances past $M_0$'s time-stamp. Events triggered by message $M_0$ may cause $P_0$ to rollback. During this process, $P_0$ may generate an anti-message $A_1$ to cancel some message $M_1$ whose destination is processor $P_1$. If $M_1$ is still in $P_0$'s send-list, then $P_0$ removes $M_1$ from the send list and discards $M_1$ and $A_1$. Otherwise, $P_0$ just discards $A_1$.

Processor $P_0$ does not need to send $A_1$ to $P_1$ to cancel $M_1$, because $P_1$ automatically cancels $M_1$ when $P_1$ processes $M_0$. Recall that each token message is broadcast to every processor. Therefore, when $P_1$ receives $K$, $P_1$ unpacks message $M_0$ and notices that $M_0$'s destination is $P_0$. Before discarding $M_0$ however, $P_1$ scans through its receive-list to check if any of the messages $P_1$ received depend on a state that $M_0$ violates. For example, if $P_0$ sent $M_1$ to $P_1$ at virtual time 45, then $M_1$ depends on the state at $P_0$ at virtual time (vt) 45 and $C_{P_1}[0] = 45$. Message $M_0$'s destination is $P_0$ ($destination(M_0) = 0$), and $M_0$ schedules an event on $P_0$ at virtual time $T_{M_0} = 43$. Since $T_{M_0} < C_{P_1}[destination(M_0)]$, processor $P_1$ cancels events scheduled by $M_1$ and removes $M_1$ from $P_1$'s receive-list.

Using Lamport clocks to track message dependencies in WTP allows a processor $P_0$ to avoid sending messages that should not be sent. For example, suppose that $P_0$ generates an event $E_a$ to be executed at processor $P_1$ at virtual time $T_{E_a} = 23$. Processor $P_0$ packs event $E_a$ with a dependency clock $C_{E_a}$ into a message $M_{E_a}$, and $P_0$ adds $M_{E_a}$ to $P_0$'s send queue. Next, $P_0$ executes an event $E_b$ scheduled by a message sent from $P_1$ at virtual time $T_{E_b} = 34$. Before executing $E_b$, processor $P_0$ updates its dependency Lamport clock so that $C_{P_1}[1] = 34$. If the next event $E_c$ generates a message $M_{E_c}$ before $P_0$ receives the token, then when $P_0$ places $M_{E_c}$ onto its send list, $P_0$ sees that $C_{E_c}[destination(M_{E_a})] > T_{E_a}$. In other words, $M_{E_c}$ depends on a state at processor $P_1$ that will be undone by message $M_{E_a}$, so $P_0$ discards $M_{E_c}$ (see Figure 2).

### 2.3 Summary of FTWP and WTP

A parallel simulation may generate messages faster than the network can deliver messages. When this happens, messages waiting to be sent accumulate in the sender's memory space, and eventually overwhelm the simulation. The FTWP and WTP protocols offer two approaches to regulating interprocessor communication in PDES. The FTWP simply forces a processor that generates messages too quickly to wait for the network to deliver the messages.

The WTP only allows processors to communicate through messages placed in the payload of a token. Only the token holder can send a token message, and every token message is broadcast to every processor. The token holder can send only one token message per possession, and the token moves between processors in a predefined order. Since every token message is broadcast to every processor, each processor can collect enough information to compute the system gvt by maintaining a Lamport clock that tracks the lvt at each processor. Finally, WTP eliminates the need for anti-message by stamping each payload message $M$ with a Lamport clock $C_M$ that tracks $M$'s dependency on the state at different processors. A processor $P$ cancels a message $M$ if $P$ sees that some state on which $M$ depends has been made invalid.

### 2.4 Reliable Broadcast over UDP/IP Multicast

The WTP is designed to function well over ethernets and other local area networks that support reliable broadcast at the physical layer. On these networks, messages passed between processors can only be lost as the result of buffer overflow at a receiving processor or a connecting network switch. However, since WTP is based on the cyclic exchange of a token, each processor can compute an upper bound on the size of its UDP receive buffer by simply placing a limit on the size of a message that a processor can send. In other words, if each processor $P$ limits its maximum message size to $B$ bytes, then $P$ can allocate a receive buffer of size $N * B$ bytes to avoid buffer overflow in an $N$ processor simulation. This simple flow control mechanism allows WTP to broadcast messages over an unloaded switched ethernet with UDP multicast without message loss.

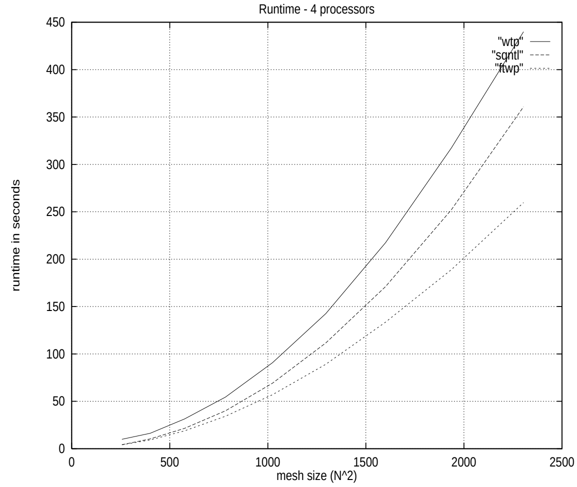
If a network switch or processor endpoint is heavily loaded, then it may sometimes lose a message despite the WTP flow control mechanism. In these environments, a processor $P_r$ that drops a message $M$ can send a negative acknowledgment (NACK) message to request that the source processor $P_s$ resend $M$. Processor $P_r$ learns that $M$ is missing when $P_r$ receives a token message from a processor that is not the token-holder, or when a timer expires.

## 3 SIMULATION PERFORMANCE

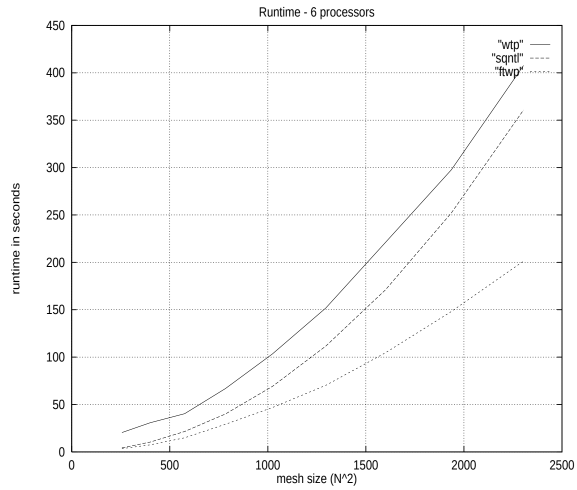
A series of simple experiments were set up to compare the performance of WTP with FTWP. Several experiments compared PARASOL's run-times using WTP and FTWP to simulate a simple baseball queueing model. The baseball consists of a mesh of $N \times N$ servers that wraps around at its ends to form a sphere. The experiment evenly distributes $N^2$ customers over the mesh of servers, and then allows each customer to move randomly between neighboring servers $N^2$ times. A customer that arrives at a server requests to be serviced for an exponentially distributed service time. If the server is busy, then the server places the customer in a FIFO queue.

Figure 4 compares PARASOL's run-times simulating the baseball benchmark with FTWP, WTP, and sequentially (on one processor) for several values of $N$. The baseball simulation runs roughly 1.75 longer with WTP than with FTWP. The measurements in Figure 5 imply that most of the difference in run time between WTP and FTWP can be attributed to the fact that each processor executes 50% more events with WTP than with FTWP to complete the same simulation. The WTP executes more events because the average rollback size of a baseball simulation is larger with WTP than with FTWP, and the average number of events between rollbacks is smaller with WTP than with FTWP.

The performance of WTP should improve if WTP's average rollback size decreases. The average rollback size would decrease if less time passed between the time when a processor executes the first incorrect event (that will be rolled back) and the time when the processor receives the straggler message that causes the rollback. One way to decrease this time is to decrease the straggler message's delivery latency. A message's delivery latency is the amount of time between




(a) 4 Processors




(b) 6 Processors

Figure 4: Baseball Runtimes with WTP and FTWP

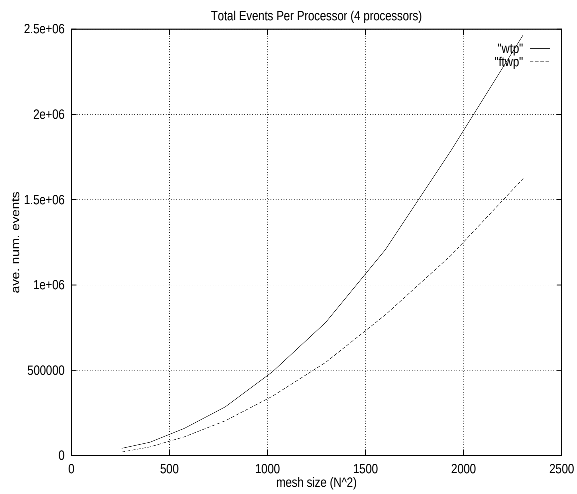



Figure 5: Total Events

when the sending processor generates the message and the receiving processor receives the message. Since the WTP requires a processor to acquire the token before sending a message, the average message delivery latency may be larger with WTP than with FTWP.

Table 1 presents average run-times for a four processor ping-pong benchmark that support the hypothesis that PARASOL's message latency is larger with WTP than with FTWP. The ping-pong benchmark begins with a single event $E_{P_0}$ on processor $P_0$ that schedules an event $E_r$ on a remote processor $P_{E_r}$ selected randomly (from $P_1$, $P_2$, or $P_3$). When $P_r$ executes event $E_{P_r}$, $E_{P_r}$ schedules another event $E_{P_0}$ on $P_0$, and the cycle repeats 1000 times. The ping-pong test is interesting because the test does not have any parallelism (only one processor is simulating an event at any give time) and the test does not involve rollbacks. The runtime of the ping-pong test completely depends upon PARASOL's ability to quickly pass events from one processor to another. The measurements in Table 1 show that PARASOL runs the ping-pong benchmark roughly 4 times faster with FTWP than with WTP.

Table 1: Ping-pong with WTP and FTWP

| Protocol | Pong Runtime in Seconds |
|---|---|
| WTP | 7.92 |
| FTWP | 1.80 |

## 4 CONCLUSIONS

The low cost and scalability of a PC and ethernet-based NOW makes it an attractive platform for PDES. Since a parallel simulation generates messages at random points in time, a message sender may not synchronize with a message receiver without risking deadlock. Therefore, PARASOL allows a processor to communicate with another processor only with non-blocking `i_send` and `i_receive` operations. Using non-blocking `i_send` bypasses the communication system's flow control mechanisms, so a sender can generate messages faster than the message passing system can deliver the messages to receivers. These messages accumulate in the sender's memory space, and eventually overwhelm the simulation.

The FTWP and WTP protocols implement two different approaches to controlling the flow of messages between processors in PDES. The FTWP simply requires a processor $P_0$ to stop simulating new events when $P_0$'s send-list grows beyond a fixed size. The WTP circulates a token between the processors participating in a parallel simulation, and a processor $P_0$ can send a message only when $P_0$ holds the token. The measurements in Section 3 show that PARASOL simulates a simple queueing benchmark in less time with FTWP than with WTP. Message delivery latency is smaller with FTWP than with WTP, so PARASOL has a shorter average rollback distance with FTWP. Since each rollback undoes fewer events, PARASOL completes a simulation in fewer total events with FTWP than with WTP.

Although FTWP has a clear advantage over WTP for the simple models presented earlier, WTP has advantages over FTWP for other models. First, WTP does not use anti-messages, so WTP should have benefits for simulations that suffer from cascading rollbacks. Models which require more than one processor to share the same simulation variables may also benefit from WTP since WTP reliably broadcasts every message to every processor. A processor can cheaply broadcast changes to a shared variable so other processors can update their cached copy of the variable. A similar mechanism may allow some models to cheaply implement distributed locks and semaphores. Exploring and expanding the range of applications where PDES can benefit simulation developers provides an unending source of future work.

## REFERENCES

Bagrodia, R. L., R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H. Y. Song. 1998. Parsec: a parallel simulation environment for complex systems. *IEEE Computer*, 31:10:77-85.

Booth, C. J. M., and D. I. Bruce. 1997. Stack-free process-oriented simulation. *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, 182-185.

Chetlur, M., N. Abu-Ghazeleh, R. Radhakrishnan, and P. A. Wilsey. 1997. Optimizing communication in time-warp simulators. *Proceedings of the 12th Workshop on Parallel and Distributed Simulation – Banff, Alberta, Canada*, 64-71.

Damani, O. P., Y. M. Wang, and V. K. Garg. 1997. Optimistic distributed simulation based on transitive dependency tracking. *Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, 90-97.

Ferscha, A. 1995. Probabilistic adaptive direct optimism control in time warp. *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, 120-129.

Gomez, J. C., V. Rego, and V. S. Sunderam. 1997. Efficient multithreaded user-space transport for network computing: Design and test of the TRAP protocol. *Journal of Parallel and Distributed Computing*, 40:1:103-117.

Jefferson, D. R. 1985. Virtual Time. *ACM Transactions on Programming Languages and System*, 7:3:404-425.

Lamport, L. 1978. Time, clocks, and the ordering of events in distributed systems. *Communications of the ACM*, 21:558-565.

Mascarenhas, E., F. Knop, and V. Rego. 1997. Minimum cost adaptive synchronization: Experiments with the ParaSol system. *Proceedings of the 1997 Winter Simulation Conference*, 389-396.

Message Passing Interface Forum. 1995. *MPI: a message-passing interface standard.*

Penesar, K. S., and R. M. Fujimoto. 1997. Adaptive flow control in time warp. *Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, 108-115.

Sunderam, V., G. Geist, J. Dongarra, and R. Manchek. 1994. The PVM concurrent computing system: evolution, experiences and trends. *Journal of Parallel & Distributed Computing*, 20:4:531-546.

## AUTHOR BIOGRAPHIES

**REUBEN PASQUINI** received his Ph.D. in Computer Sciences from Purdue University in 1999. His research interests include parallel simulation and distributed systems.

**VERNON REGO** is a Professor of Computer Sciences at Purdue University. He received his M.Sc.(Hons) in Mathematics from B.I.T.S (Pilani, India), and an M.S. and Ph.D. in Computer Science from Michigan State University (East Lansing) in 1985. He was awarded the 1992 IEEE/Gordon Bell Prize in parallel processing research, and is an Editor of *IEEE Transactions on Computers*. His research interests include parallel simulation, parallel processing, modeling and software engineering.