Seven-O'Clock: A New Distributed GVT Algorithm Using Network Atomic Operations

David Bauer, Garrett Yaun, Christopher D. Carothers
Murat Yuksel and Shivkumar Kalyanaraman
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, U.S.A.
{bauerd,yaung,chrisc}@cs.rpi.edu, {yuksem,shivkuma}@ecse.rpi.edu

Abstract

In this paper we introduce a new concept, *network atomic operations (NAOs)* to create a zero-cost consistent cut. Using NAOs, we define a wall-clock-time driven GVT algorithm called *Seven O'Clock* that is an extension of Fujimoto's shared memory GVT algorithm. Using this new GVT algorithm, we report good optimistic parallel performance on a cluster of state-of-the-art Itanium-II quad processor systems for both benchmark applications such as PHOLD and real-world applications such as a large-scale TCP/Internet model. In some cases, super-linear speedup is observed.

1 Introduction

At the heart of an optimistic parallel simulation system is the ability to reclaim memory and re-use it to schedule future events as well as support state-saving operations as part of speculative event processing. Global virtual time (GVT) defines a lower bound on any unprocessed event in the system and defines the point beyond which events *should not* be reclaimed. Thus, it is imperative that the GVT computation operate as efficiently as possible.

Global virtual time (GVT) algorithms must solve two key problems. The first is *the transient message problem*. Here, a message is delayed in the network and neither the sender nor the receiver consider that message in their respective GVT calculation. Thus, a GVT algorithm must account somehow for all messages scheduled. The second problem is called *simultaneous reporting*. This problem arises "because not all processors will report their local minimum at precisely the same instant in wall-clock time" [8]. Here, the underlying assumption is that event processing is allowed to continue asynchronously during the GVT computation which enables better overall parallel performance.

Asynchronous GVT algorithms rely on the ability to create a "cut" across the distributed simulation that divides events into two categories: past and future [8, 18]. GVT is then defined by the lower bound on unprocessed events in the "past" of the cut, which is in effect an estimate of the true GVT since events are being processed during its computations. Creating a cut can be done in several ways depending on the architecture of the machine(s) being used. For distributed computing platforms, the primary method of creating a "cut" is via message-passing as was defined by Mattern [18]. Here, messages are sent such that at most two cuts are made. The first cut signals the "start" of the GVT computation. The second cut, if needed based on message counts computed in the first cut, consider any transient messages discovered from the first cut. Please note, that these cuts need not be *consistent*. A *consistent cut* is defined as a cut where there is no message that was scheduled in the future of the sending processor but received in the past of the destination processor. These messages can be ignored because by definition they must be scheduled at a time that is greater than GVT computed using a consistent cut. The "cuts", while not consistent, effectively divide past from future in a causally consistent manner to solve the simultaneous reporting problem. Additionally, because of the use of message counts, it is able to determine if a second "cut" round is needed and traps any transient messages.

In contrast, a shared memory multiprocessor greatly simplifies the GVT algorithm. Fujimoto's GVT algorithm [7] generates a cut by setting a global flag positive. In a shared memory system this operation is observed on all processors in a causally correct order because the underlying hardware memory management mechanism ensures that no two processors will observe different orderings of memory references to a shared variable. Shared memory multiprocessor systems that adhere to this memory ordering model are called *sequentially consistent* [14]. The impact this memory model has on a GVT algorithm are that: (i) *no* messages are lost which prevents the transient message problem, and (b) the simultaneous reporting problem is solved because all processor effectively "observe" the start of the GVT calculation at the same instant of wall-clock time.

The motivation behind our research here is the question: *Is there a method to achieve some of the benefits of sequentially consistent shared memory but in a loosely coordinated, cluster computing environment?* The answer turns out to be *yes*. In this paper, we propose the idea of a *network atomic operation (NAO)*, which enables a zero-cost cut mechanism which greatly simplifies GVT computations in a cluster com-

Algorithm 1 Fujimoto's Shared Memory GVT Algorithm: Variable Definitions.

Global Variables

```
int gvt_flag;
lock_t gvt_lck; /* mutual exclusion variable */
int gvt_interval; /* number of time thru batch loop */
virtual_time_t lvt[npe]; /* LVT of each processor */
virtual_time_t gvt;
```

Processor Private Variables

```
virtual_time_t send_min_ts;
```

Algorithm 2 Fujimoto's Shared Memory GVT Algorithm: Initiate GVT.

Steps to Initiate GVT within the Scheduler Loop

```
if(this processor is the MASTER)
{
  gvt_cnt++;
  if( gvt_cnt >= gvt_interval AND
  processor status is GVT_NORMAL)
  {
    gvt_cnt = 0;
    if(gvt_flag == -npe)
    {
      lock(&gvt_lck);
      gvt_flag = npe;
      unlock(&gvt_lck);
    }
    set processor status to GVT_COMPUTE;
  }

}
else if (gvt_flag > 0 AND
         processor status is GVT_NORMAL)
  set processor status to GVT_COMPUTE;
```

Algorithm 3 Fujimoto's Shared Memory GVT Algorithm: Receive Events.

Steps to Receive Events and Anti-messages within Scheduler Loop

```
move positive messages from shared memory
message queue to processors priority queue.
process any rollbacks.
remove anti-messages from shared memory
``cancel'' queue.
process any message cancellations and
rollbacks.
```

puting environment. We demonstrate its reduced complexity by extending Fujimoto's shared memory algorithm to operate across a cluster of shared-memory multiprocessors (SMP). We present a performance study using the PHOLD benchmark and TCP network model.

Algorithm 4 Fujimoto's Shared Memory GVT Algorithm: Compute GVT.

Steps to Compute GVT Once Initiated within Scheduler Loop

```
if( processor status is GVT_COMPUTE )
{
  set processor status to GVT_WAIT
  lvt[my_pe] = min(send_min_ts,
                   smallest event in
                   priority queue);
  lock(&gvt_lck);
  gvt_flag--;
  if( gvt_flag == 0)
  {
    gvt = min( lvt[0] ... lvt[npe-1] );
    gvt_flag = -1;
    set processor status to GVT_NORMAL;
    reset send_min_ts to max time value;
    unlock(&gvt_lck);
    collect processed events and state < GVT;
    if( gvt > end time of simulation )
              goto DONE;
  } else
  {
    unlock(&gvt_lck);
    set processor status to GVT_WAIT;
  }
}
else if(processor status is GVT_WAIT AND
                              gvt_flag < 0 )
{
  lock(&gvt_lck);
  gvt_flag--;
  unlock(&gvt_lck);
  if( gvt > end time of simulation )
    goto DONE;
  collect processed events and state < GVT;
  set processor status to GVT_NORMAL;
  reset send_min_ts to max time value;
}
```

2 Fujimoto's GVT Algorithm and NAOs

We begin with an overview of Fujimoto's GVT algorithm, as shown in Algorithms 1 – 5. Please note, there are minor modifications from the original algorithm presented in [7], but the correctness and efficient execution is preserved. These 5 parts are described as follows:

1. **Variables (Algorithm 1):** The key shared variable is the `gvt_flag`, which contains a mutual exclusion variable, `gvt_lck`.
2. **Initiate GVT (Algorithm 2):** The algorithm is initiated when the "master" processor iterates through the event scheduler loop `gvt_interval` times before setting the `gvt_flag` equal to the number of processors

Algorithm 5 Fujimoto's Shared Memory GVT Algorithm: Process and Schedule Events.

Steps to Process and Schedule Events within Scheduler Loop

```
  Process smallest event in priority queue;
  if( sending a new event during
                       event processing )
  {
    enqueue message on destination
                receive queue;
    if( gvt_flag > 0 AND
         processor status is NOT GVT_WAIT )
        send_min_ts = min( send_min_ts,
                  time-stamp of new event);
  }

DONE:
compute parallel simulation stats and exit.
```

(i.e., npe). Here is where the algorithm exploits the sequentially consistent memory properties. Every processor will "observe" the start of the GVT at the same instant in wall-clock time. More precisely, once the flag has been set, any other messages sent, are the responsibility of the sender, as shown in Algorithm 5. This provides a true separation between events in the logical past and logical future. When another processor observes the start of a GVT computation, it changes its status to "needs to compute an local virtual time (LVT)" This is denoted by `send_min_ts`.

3. **Receive Events (Algorithm 3):** Here, new events and anti-messages are processed from arrival queues shared between processors. Each processor has its own externally exported queue that all other processors use to send events. A mutual exclusion lock is used around the queue to correctly serialize the arrival of either new events or anti-messages. Because of sequentially consistent memory, no message can be lost "in the network" and so the transient message problem is intrinsically solved. Observe that this "receive" and process rollbacks and anti-messages is a necessary step prior to computing any part of the GVT. It is also a normal step in every iteration through the scheduler loop.

4. **Compute GVT (Algorithm 4):** Once the new events and anti-messages are processed, each processor computes its local virtual time (LVT) value, which is the smallest unprocessed event that it is "aware" of, which includes any events it sent after the `gvt_flag` was set. This is denoted by `send_min_ts`. The last processor to compute its LVT also computes the minimum among all LVTs, which becomes the new GVT value. To inform other processors that the new GVT value is available, the `gvt_flag` is set to negative one. The last processor never "waits" for the GVT value and skips that state, while all other processors move to the "asynchronously waiting for GVT" state.

Algorithm 6 `gvt_interval` is a predefined number of iterations, and gvt_count is the current number of iterations.

```
CPU 0:
   a. If(gvt_interval == gvt_count)
   b.        set gvt_flag positive

CPU 1:
   c. if(gvt_flag)
   d.      start processing LVT
```

5. **Process Events (Algorithm 5):** In this last step, forward event processing commences. Here, the smallest event is removed from the pending event set and processed. If a new event is scheduled and the `gvt_flag` has been set greater than zero and the LVT value has not been reported (i.e., the processor should not be in the "wait" state), then it means *this* processor must consider this event in its LVT computation.

2.1 Network Atomic Operations

To directly extend Fujimoto's GVT algorithm to a network of machines, would require a sequentially consistent distributed memory model, similar to what is provided in a shared memory system. As we described above, each processor observes the "start" of the GVT computation at the same wall clock time because of sequentially consistent memory. In reality, a processor attempting to read the flag may be stalled while the underlying system updates the local cache with the correct value of the flag. Consider the following abstracted view of the algorithm run in parallel on an SMP machine as shown in Algorithm 6.

Running this code on a single processor defines the sequential consistency. The statements could be ordered on a single CPU as $O = \{a, b, c, d\}$. In this case both CPU 0 and 1 would begin computing GVT. However, if we change the interleaving of instructions to $O = \{a, c, b, d\}$ then the GVT computation would only begin on CPU 0. CPU 1 would begin processing more events, but when an event is sent, the `gvt_flag` would be checked per the algorithm to ensure the sender correctly accounts for events during the GVT computation. The instruction, call it e, would then be accounted for by CPU 1 because e occurred after instruction b, and the consistent cut is properly formed.

NAOs provide a similar functionality in a distributed system, however they are clock-based and not memory or state-based. **The general concept is that an operation may occur atomically within a network of machines if all machines "observe" the event at the same instant of wall clock time.** This functionality can be implemented on modern processors because most architectures now provide a time-stamp counter, or clock-cycle counter for performance measuring, such as the `rdtsc` instruction on all x86 series processors [11]. So we can compute wall-clock time based off of each processor's time-stamp counter and synchronize these counters to a common view of wall clock time. Calls to read-

Algorithm 7 Here, `gvt_interval` redefined as a measure of time usually in clock cycles, and not defined as the number of batch round through the scheduler.

```
CPU 0:
   a. if(local clock time >= gvt_interval)
   b.       start computing GVT

CPU 1:
   c. if(local clock time >= gvt_interval)
   d.     start computing GVT
```

ing the CPU clock adhere to the principles of a sequentially consistent memory model because wall clock time is consistent across all processors. Consider the following clock-based approach shown in Algorithm 7.

If we again attempt to create a sequential ordering of instructions, it becomes obvious that any permutation is guaranteed to be consistent. Consistency is guaranteed because instructions a and c in Algorithm 7 will evaluate to true if and only if the same instance of wall clock time has passed for each CPU. Because we can only read the current wall-clock time (as measured in clock cycles), any permutation of the possible orderings is valid because wall clock time is *assumed* to be the same for all processors. There are limitations which we will discuss later in the paper.

So NAOs may be characterized as a subset of the possible operations provided by a complete sequentially consistent memory model. For example, NAOs can only occur at predefined intervals, not dispersed throughout the time-scale of the running application. Not only must NAOs occur at agreed upon intervals, but they must also take on a specific meaning or value. In the case of the GVT algorithm, we use NAOs to generate a consistent cut. The system is either in a GVT computation, or it is not. Further, GVT computations occur at a predefined frequency through the runtime of the application. An NAO cannot be used to give some global variable any value, because the only global variable in an NAO is wall clock time. However, any sequence of operations can be performed once the clock has been read.

2.2 Clock Synchronization

The heart of a network atomic operation is the assumption that all processors share a highly accurate, common view of wall-clock time. For this to occur, each processor's time-stamp or cycle-counter must be synchronized in some fashion. This is a well researched problem in distributed computing. The most recent, relevant result for our operating environment is by Ostrovsky and Patt-Shamir [20]. Here, they present provably optimal clock synchronization scheme where the clocks have drift and the message latency may be unbounded. Previous to this result, all other optimal results were based on non-drifting clocks. Moreover, they suggest that operational clock synchronization algorithms *need not be general* and "that they should work for the particular system in which they are deployed". We take this view here. In particular, because of the time-scale of the clock is 1,000 times greater than message sends (i.e., nanoseconds vs. microseconds), clock drift rates can largely be ignored here.

Additionally, since our contribution is not about clock synchronization algorithms, we used a simplified approach. To synchronize the clocks across all processors, we use a *network barrier*. Here, a master time keeper sends a synchronization message to each node, which responds back with its local time-stamp-counter. The master time keeper then sends a message to each processor with an appropriate time-stamp counter value that would be when in real-time measure in cycles the first GVT is to occur. Upon receipt of that message, each processor is released from the barrier and begins processing events. We recognize that for a large 1000 processor cluster, this approach has some scalability limitations. For such an operating environment, we would implement Ostrovsky and Patt-Samir algorithm [20].

3 Seven O'Clock GVT Algorithm

We can now give a definition of a network atomic operation:

Definition: *An NAO is an agreed upon frequency in wall-clock time at which some event is logically observed to have happened across a distributed system.*

Each processor in the system uses the NAO to determine the current state of the system depending upon the logical meaning of the NAO. For example, we use an NAO to determine if a GVT computation has been started. There is no actual global variable, such as the `gvt_flag` in Algorithm 2, which signals the start of the GVT. Instead, we simply compute GVT every n units of wall-clock time.

We have affectionately call this algorithm the *"Seven O'clock Algorithm"*. Seven O'clock comes from the idea that if we could have synchronized wall clocks on each network node, then we could simply compute GVT at well-defined intervals, i.e., every minute starting from Seven O'Clock, where Seven O'Clock is simply the start time of the scheduler. During the discussion of this algorithm we assume that each processor's timestamp counter is perfectly synchronized with all of the other counters. We introduce the complexity of clock drift and jitter at the end.

As we previously indicated, if one were to open up the underlying hardware implementation of a sequentially consistent shared memory system, a number of messages would be observed being passed over the memory bus between memory and cache modules. Also, any memory reads to a shared location could be blocked while waiting for the memory address to be made consistent. In particular, these consistency messages would either be well synchronized in time over a memory bus or acknowledged over a network depending on the architecture [9, 15]. Because we assume a distributed message passing system without acknowledgments, we need some additional information about the communications environment to avoid the transient message problem (i.e., events lost in the network). This problem can be overcome by adding a small amount of time to the NAO expiration, $max_send_delta_t$.

Definition: *$max_send_delta_t$ is a worst case bound on the time to send an event through the network.*

This delta allows for the sending processor to account for remotely sent events which may cross the cut boundary. Note that this is not the same as delta causality which allows for excessively old events to be discarded. While it may seem unreasonable to assume such a value can be determined in practice, current cluster computing networks rely on high-speed switching fabrics. These fabrics typically have extremely low loss probabilities ($1e-12$ or less) and can typically support the full bandwidth of all ports. Consequently, the worst case is experimentally computable and does not vary greatly from the average case.

The states of the Seven O'Clock algorithm are shown in Figure 1 and are discussed below.

- **State A:** Events are processed normally and not accounted for in GVT computation. This is no different from Fujimoto's GVT Algorithm.
- **GVT "start":** The NAO signals the consistent start of the GVT to all processors, just as setting the `gvt_flag` in Fujimoto's algorithm.
- **State B:** Events sent during the `max_send_delta_t` interval are accounted for on the sending side. This is similar to how Fujimoto's Algorithm uses `gvt_flag` to capture events the receiving processor might not consider in the sender's LVT computation. Here, our algorithm reads the processor's local cycle counter, determines if it is within the `max_send_delta_t time` of a GVT computation. If it is, then it captures this event's timestamp against the minimum of any previously scheduled events during the `max_send_delta_t` time.
- **State C:** Processors compute LVT by taking min(unprocessed events, events sent during B). This is identical to Fujimoto's GVT Algorithm.
- **State D:** Node 1 is first to complete local GVT algorithm and propagates this value to other nodes.
- **State E:** Node 0 completes it's local GVT algorithm and receives Node 1's. It takes the minimum of the two and retransmits this value back to Node 1. The other processors check for the new GVT value at some point in the future and read the value from the shared memory. The processors return to state A.

The only other differences between Fujimoto's GVT algorithm and Seven-O'Clock are: (a) physically receiving messages and (b) physically sending messages. In the receiving step (Algorithm 3), all remotely sent new event messages and anti-messages are read from a communications channel, such as a socket. This is in addition to the shared memory queuing structures. From the standpoint of the GVT algorithm, it captures messages sent over either communications medium. Likewise, when a message is sent, the algorithm does not differentiate between which events are shared memory or off-system messages.

3.1 Proof of Correctness

As previously noted, in order for a GVT algorithm to operate correctly, it must solve the transient message and simultaneous reporting problems.

First, we assume all processors clocks are perfectly synchronized and there are no clock drift or jitter problems. We will relax this constraint later.

Proof: To prove that a transient message cannot occur, assume that a transient message occurs in the scheduler. Then the time to send the event must be greater than the $max_send_delta_t$. But by definition $max_send_delta_t$ is a worst-case bound on the time to send an event. This leads to a contradiction because the transient event took longer to send than the worst-case bound.

Next, the simultaneous reporting problem occurs when all processors do not begin computing their local minimum at the same instant. In fact, this is exactly what happens in the Seven O'Clock algorithm. Because the consistent cut is generated using an NAO, each processor in the distributed system begins accounting for messages at precisely the same instant in wall clock time. Therefore this problem does not occur in this GVT algorithm.

Theorem: *The simultaneous reporting problem cannot occur in a system where a consistent cut is defined across all processors at precisely the same instant in wall clock time.*

Proof: *Assume to the contrary that the simultaneous reporting problem can occur.*

CASE 1: Assume each processor's clock is perfectly synchronized with all other processor clocks. For the simultaneous reporting problem to occur, at least two processors must have different views of wall clock time. This is a contradiction because there only exists one notion of wall clock time.

CASE 2: Each processor's clock is synchronized with some degree of error, which is bounded by $epsilon$. In order for the problem to occur now, at least two processors must have different views of wall clock time, which differs by at most $epsilon$. This means that in $epsilon$ time between two processors, a message was sent which was not accounted for. This is a contradiction because it is not possible to send a message in $epsilon$ time and not account for the event being sent. Consider the steps for remote sending of events:

1. *Send the event.*
2. *Read local time-stamp clock.*
3. *If $time_now + max_send_delta_t >= gvt_interval$, then account for the event.*

Since $epsilon$ is a value far less than $max_send_delta_t$, we must always account for the sent event. In the event that $epsilon >= max_send_delta_t$, we simply change our $max_send_delta_t$ to have a larger value to overcome $epsilon$.

3.2 Problems with Clock-Based Algorithms

While the discussion of clock synchronization and its associated problems are outside of the scope of this paper, the Seven

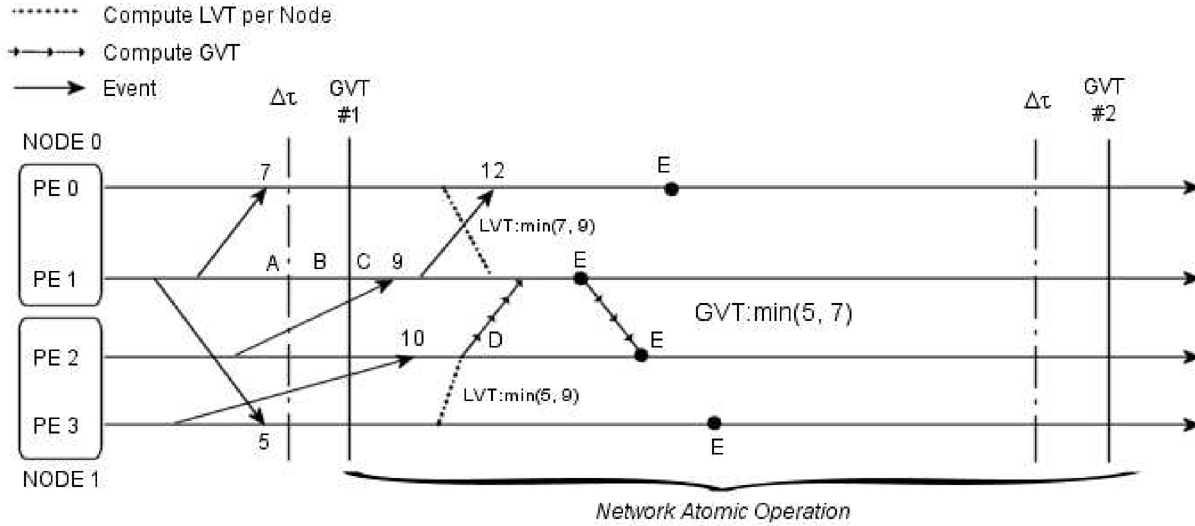

Figure 1: States of the Seven O'Clock Algorithm.

O'Clock algorithm does provide a mechanism to solve each of these problems. Three problems arise in any clock-based algorithm: drift, jitter and synchronization error. During the course of a simulation, clocks may drift together, or apart. In the later case, this can lead to a gradual disparate view of time. Clock jitter occurs when a time is discretized and the width of the time units is not uniform. This can be a cause of clock drift over time depending on the frequency and size of the jitter. Finally, it is difficult to synchronize clocks to a high degree of granularity. This can lead to two processors being synchronized, but off by an indeterminate amount. Each of these problems can be dealt with in the Seven O'Clock algorithm by adjusting the definition of $max_send_delta_t$. Recall that this value is a worst-case bound on the time to send an event between two processors.

We now redefine max_send_delta_t as the maximum of:

1. worst-case bound on events sends
2. two times the synchronization error
3. two times the maximum clock drift during execution

We observe that the $max_send_delta_t$ parameter is simply an adjustment to the NAO to compensate for in-transit events that the sender must account for if they will not be received prior to the NAO expiring at the receiver. The "two times" factor comes from the observation that one processor could have a max synchronization error or clock drift in the negative direction relative to the absolute real time (i.e., master node) and the other processor could have the max synchronization error or clock drift in the positive direction. Thus, these two processors are "out of synch" by at most a factor of two times maximum error or drift. However, in practice these error and drift values are several orders of magnitude smaller than the maximum time to send an event through the network, and can be safely ignored in most operating circumstances.

3.3 Limitations

Two issues arise from the introduction of the Seven O'Clock algorithm that do not exist in other GVT algorithms. Both stem from the general problem which is that the Seven O'Clock GVT algorithm cannot be "forced". First, GVT must advance for a simulation to determine that the simulation must end. In the *Rensselaer's Optimistic Simulation System*(*ROSS*) parallel scheduler this happens quickly because events to be scheduled past the end time are not processed and the GVT interval counter climbs quickly so that GVT may be computed when the system is effectively out of events. The Seven O'Clock algorithm cannot be forced because each node must wait for the NAO to expire. At the end of a simulation, the *ROSS* system has no events scheduled for processing, and is simply waiting for the CPU to compute the next GVT interval. This situation cannot be aborted early because each network node is unaware of other nodes still actively processing events. The time wasted is bounded in the worst case by the size of the GVT interval. It is reasonable to expect this value to be small in relation to the overall time spent in execution, and so we do not consider this to be a major loss because it can be effectively amortized away.

The fact we cannot force a GVT computation leads to the second limitation to the system. When all of the free events were consumed in the *ROSS* parallel scheduler, we were previously able to "jump" the GVT interval counter and force a GVT computation to occur. Refreshing the GVT value meant that we could reclaim at least one additional event in the system and continue forward processing. When free events are exhausted in the *ROSS* distributed scheduler we can no longer force GVT, and so must simply wait for the next GVT interval to pass. This problem occurs when we do not have sufficient optimistic memory to continue forward execution. The solution to this problem is to simply add more optimistic memory, *or more network nodes*, further distributing the model so that this does not occur. An indirect cause of this problem is spec-

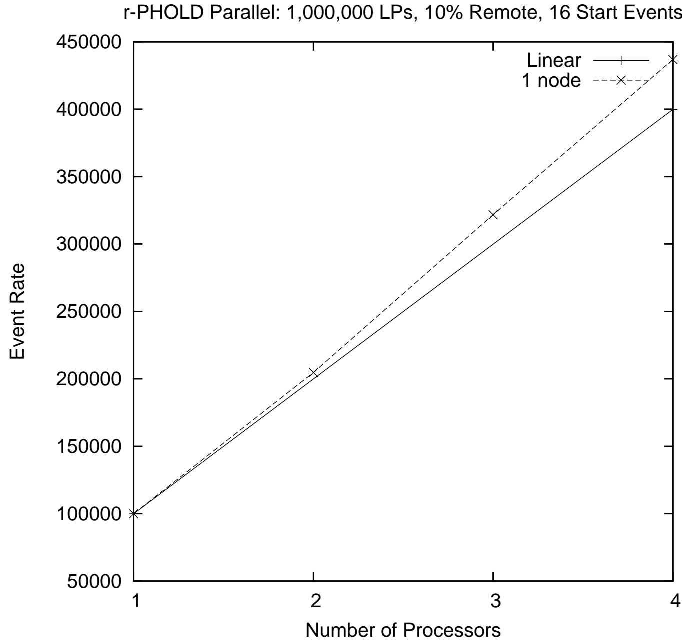

Figure 2: PHOLD results in parallel case.

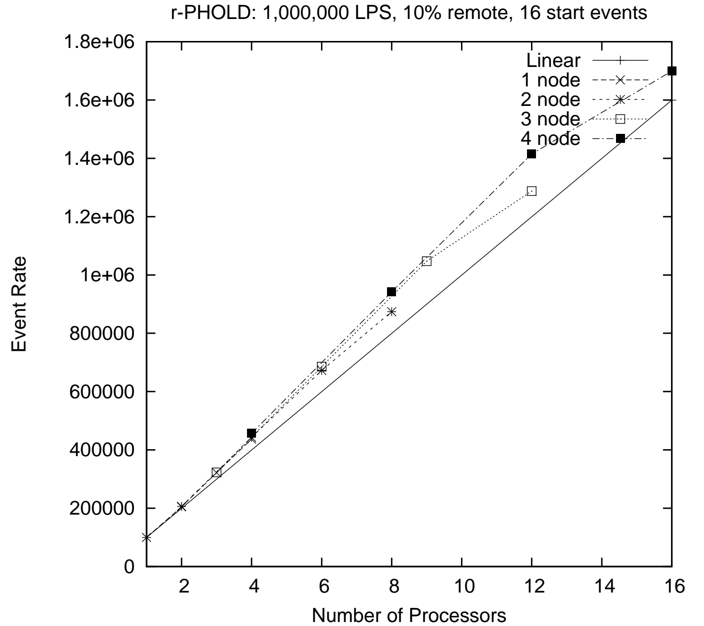

Figure 3: PHOLD results in the distributed cases with 1,2 or 3 processors utilized per node. The 4 processor cases are clearly affected by context-switching in the operating system for I/O operations.

ulative execution. In this case, stalled waiting for GVT to pass can act as a throttle on the faster nodes in the network such that they cannot overly speculatively execute, thereby creating the potential for long rollbacks. This problem is best solved by tuning the GVT interval to more closely match the amount of available memory.

4 Performance Study

There are two benchmark models used in this performance study. The first is a synthetic workload model called PHOLD. This commonly used benchmark has been modified to support reverse-computation and is configured to have minimal LP state, message sizes and event processing. The forward computation of events involves computing three random numbers: one for computing if a remote event should be created, one used to compute the time-stamp and one used for the destination LP. The reverse computation involves "undoing" an LP's random number generator (RNG) in order to restore it's state. Because the RNG is perfectly reversible, the reverse computation restores seed state by computing the perfect inverse function as described in [3]. The destination LP is determined by calling a uniformly distributed random number generator in the range of 0 to 100. If the generated value is less than the specified percent of remote messages allowed, we choose the destination LP over an exponential distribution of LPs, otherwise, the event will be sent to the source LP. The third call determines the offset timestamp for events and is exponentially distributed with a mean of 1.0, with the model completing at timestamp 100. For all experiment runs, we mapped LPs to PEs in a round robin fashion. Each simulation run contained 1 million LPs, and the number of KPs was

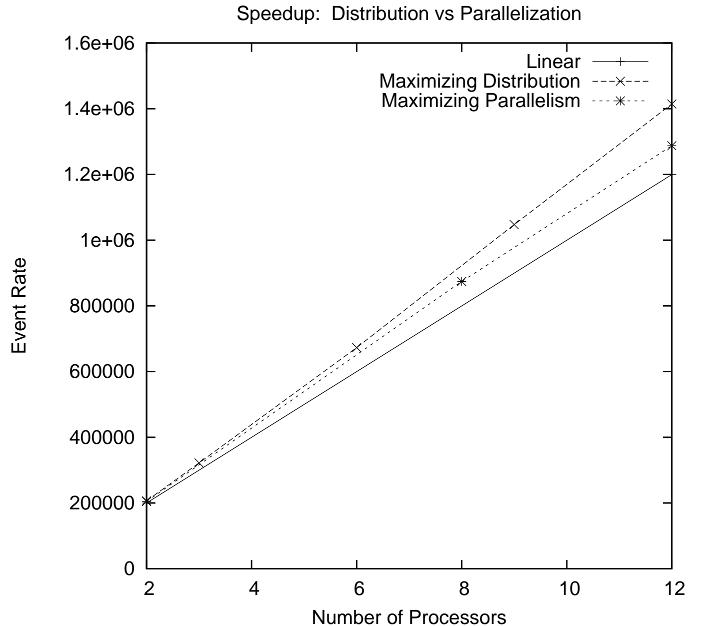

Figure 4: Maximizing nodes used generates better speedup than maximizing processors per node.

determined as $NPEs^2 * 4$. The message population per LP is 16. This model is a pathological benchmark which has minimal event granularity while producing a configurable number of remote events which can result in "thrashing" rollbacks.

The second application is a model of TCP and this imple-

mentation follows the Tahoe specification [5]. There are three main data structures in this model: the data packet which is sent between hosts in the data plane, the network router LPs which maintain queuing information and forward packets through the network and the host LPs which keep statistical information on the transferring of data. For detailed model design and implementation, we refer the interested reader to [25].

For all TCP experiments, each TCP connection maintained a consistent configuration. The transfer sizes are infinite and so each TCP sender/receiver pair operates for the duration of the simulation. Each host pair is generated randomly during the initialization of the model. The random nature of connections means that there is a high percentage of "long haul" links which results in a large number of remote events being scheduled between PEs. The synthetic topology is hierarchical, and contains 4 levels, the top-most being fully connected. A router in a given level has N lower level routers or hosts connected to it, so the total number of nodes in the system is equal to: $N^4 + N^3 + N^2 + N$. The nodes were enumerated in such a way that the next hop can be calculated based on the destination at each hop. The bandwidth, delay and buffer size for the synthetic topology is as follows:

- 2.48 Gb/s, a delay of 30 ms, 3MB buffer
- 620 Mb/s, a delay between 10 and 30 ms, 750 KB buffer
- 155 Mb/s, a delay of 5, 1o and 30 ms, and 200 KB buffer
- 45 Mb/s, a delay of 5 ms, and 60 KB buffer
- 1.5 Mb/s, a delay of 5 ms, and 20 KB buffer
- 500 Kb/s, a delay of 5 ms, and a 15 KB buffer

4.1 Computing Testbed and Experiment Setup

The Itanium2 processor [10] is a 64 bit architecture based on *Explicitly Parallel Computing (EPIC)* which intelligently bundles instructions together that are free of data, branch or control hazards. This approach enables up to 48 instructions to be in flight at any point in time. Current implementations employ a 6-wide, 8-stage deep pipeline. A single system can physically address up to 2^{50} bytes and has a full 64-bit virtual address capability. The L-3 cache comes in a 3 MBs configuration and can be accessed at 48 GBs/second which is the core bus speed. For all experiments here, up to 4, quad Itanium processor servers were use. TCP over Gigabit Ethernet was used as the interconnection network between quad processor systems.

When applicable, the parallel scheduler computed 64 events per batch loop, and computed GVT after 64 batch loops. Thus, up to 4096 events will be processed between GVT epochs. These settings have been determined in the past to yield high performance. For the Seven O'Clock Algorithm, the NAO is configured to fire every 250 milliseconds to compute GVT. Optimistic event memory was computed in each case from the following formula: $OptimisticMemory = C * NumPEs * Batch * GVT_{Interval}$ [3]. Here, $NumPEs$ is the number of processors used within the network, and C is a small constant factor. Each processor can consume $Batch * GVT_{Interval}$ events per GVT epoch and C determines an amount of reserve buffers for optimistic execution during the asynchronous GVT computations. During distributed execution, each processor can consume approximately the NAO time interval (in cycles) divided by the average time (in cycles) to compute one event per GVT epoch.

4.2 PHOLD Performance Data

For the Itanium cluster results, we configured PHOLD with 10% remote messages with 16 seed event per LP and used the Myrinet network. Shown for completeness, Figure 2 shows that *ROSS* parallel scheduler continues to provide linear speedup. In fact, it proved difficult to derive a sequential case which did not show a slight super-linear speedup. Comparing the results in the parallel and distributed case in Figure 3 is complicated because we have results based both on the number of processors used and on the number of nodes used. We start by considering two nodes maximizing the processors before increasing the number of nodes. Then we consider three nodes, and finally four nodes.

When a fourth processor is allocated per node, performance degrades. We attribute this phenomenon to our simulation processes competing with other operating system service processes resident on each node, such as Network File System (NFS) daemons, for processing time. For two and three processors cases, the Linux scheduler would bind those service jobs to other available processors. When all four processors are used, the Linux scheduler now must multiplex our simulators' threads and these service processes among all processors leading to an increase in context switch overheads. To confirm this hypothesis, we measured the amount of context switches which occurred in each case. For the two and three processor cases we observed 12 and 14 involuntary context switches respectively. However, for the four processor case, we found 1123 involuntary context switches. This is a 100 fold increase.

After close analysis of the parallel and distributed system we conclude that there are two possible speedups in a parallel and distributed system: speedup due to parallelism and speedup due to distribution. In Figure 4 we see that the system is in fact generating a better speedup as we go more distributed. Besides context switching, other problems may arise such as memory bus overloading or serialized memory references [2, 15, 19], which limit the possible speedup due to parallelism. By maximizing the number of nodes used in a simulation it is possible to avoid or reduce these problems by de-coupling the hardware systems which are limiting performance.

4.3 TCP Performance Data

Computing the LP to PE mapping for the TCP-TAHOE model becomes difficult when the number of processors available is not some multiple of 32. Therefore we present here the results for the 2- and 4-node simulation runs. Figure 5 shows that we again generate a very linear speedup. However, the speedup is sub-linear because of the available num-

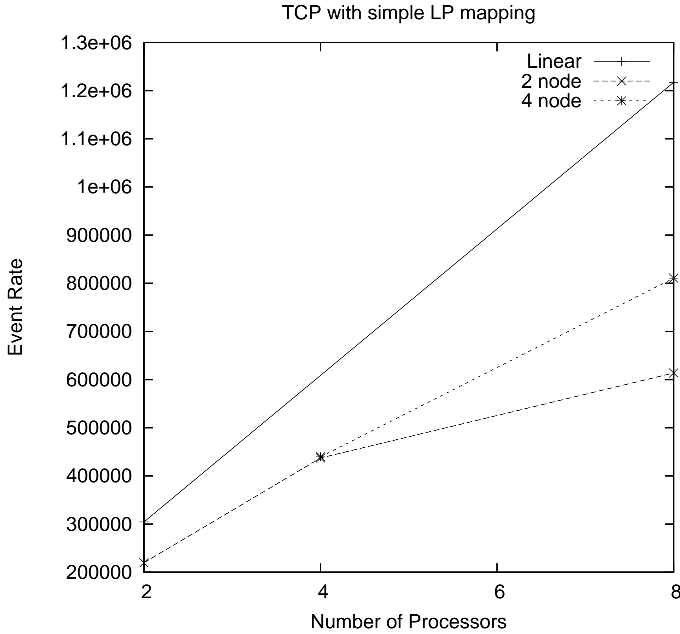

Figure 5: Results for TCP-TAHOE running on 2 and 4-nodes.

ber of kernel processes (KPs). In [3], KPs where introduced as an aggregation structure for reducing LP fossil collection and rollback overheads. In our implementation of this optimization, we hypothesize that we are seeing the same performance problem here, but on a much larger scale. With 1 million LPs, each modeling some end (either source or destination) of a TCP connection, the model requires far greater than 32 KPs in order to achieve good performance. So it is constrained by the way the topology of the TCP network was mapped to LPs/KPs/PEs. However, we believe with a better mapping, the performance would improve.

5 Related Work

Samadi's algorithm [24] solves the simultaneous reporting problem through the use of message acknowledgments. Here, processors will tag any "ack" message that it sends in the period from when the processor reports its LVT until it receives the new GVT value. This process prevents any messages from "slipping through the cracks" [8].

In addition to Fujimoto's and Mattern's GVT algorithms, there have been a number of predecessors. Preiss [22] introduced a scheme which places the PEs into a ring. The first round completes when a token has been passed around the ring and returns to the initiating PE. When the initial token is received, a PE begins accounting for remotely sent messages which may or may not complete prior to the GVT computation. On receiving the second token, a local GVT value is computed as the minimum between the value stored in the token and the PEs local minimum. Improvements to this algorithm have been proposed by Bellenot [1] which reduce the complexity of message passing by organizing PEs into trees rather than a ring. Lin and Lazowska [16] propose a new data structure that reduces the frequency of acknowledgment messages. The work done by Tomlinson and Garg [23] uses counters to detect transient messages. However, this scheme does not employ the use of a "cut" as Mattern's algorithm does. Pancerella [21] propose a hardware based scheme whereby host systems using custom network interface cards are interconnected to form an efficient reduction network to rapidly compute GVT.

In follow-up research, Lin [17] uses control messages to efficiently find/flush out transient messages. At about the same time, D'Souza et. al [4], proposes a statistical approach to estimating GVT.

Prior to Fujimoto's GVT algorithm, Xiao et al. [26], proposes an asynchronous GVT algorithm that exploits shared-memory multiprocessor architectures. Here the concept of "cages" is used, where each processor own some set of "cages" that it places or lays down in order to track the LVT of that processor. It is interesting to note, that here a global *Cage Flag* much like the `gvt_flag` in Fujimoto's Algorithm, is used to "kick off" the GVT computation.

Related to both "consistent cuts" and sequentially consistent memory is the issue of causality. Most recently, Zhou et al [27], propose a "relaxed" causal receive ordering algorithm called *critical causality* for distributed virtual environments.

6 Conclusions

We present a new idea of Network Atomic Operations, and apply it to solving the global virtual time problem. We use an NAO to generate a zero-cost cut in a distributed system which is both scalable and efficient. Our experimental results indicate a super-linear speedup over sequential. We also propose that there are two forms of speedup, one due to parallelization and one due to distribution. The parallel scheduler results were already super-linear, however there is more speedup inherent to distribution because we eliminate bottlenecks such as the memory bus and context switching. Finally, NAOs lead us to viewing the behavior of Time Warp in the frequency domain because they define the relationship between virtual time and wall-clock time. In the future, such a view may potentially prove beneficial in better understanding Time Warp behavior.

Acknowledgments

This research is supported by NSF CAREER Award CCR-0133488, NSF CNS Award CNS-0435259, DARPA Network Modeling and Simulation (NMS) program, contract #F30602-00-2-0537 and a AT&T University Relations Program Grant.

References

[1] S. Bellenot, "Global Virtual Time Algorithms", In *Proceedings of the SCS Multiconference on Distributed Simulation*, Vol. 22, pp. 122–127, 1990.

[2] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto, “The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture,” in *Proceedings of the 1999 Winter Simulation Conference*, 1999.

[3] C. D. Carothers, D. Bauer, and S. Pearce, “Ross: A high-performance, low memory, modular time warp system,” *Journal of Parallel and Distributed Computing*, 2002.

[4] L. M. D’Souza, X. Fan, and P. A. Wilsey, “pGVT: An Algorithm for Accurate GVT Estimation”, In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS ’94)*, pp. 102–109, 1994.

[5] K. Fall and S. Floyd, “Simulation-based comparison of tahoe, reno, and sack tcp,” *Computer Communication Review*, vol. 26, pp. 5–21, 1996.

[6] R. M. Fujimoto, “Time warp on a shared memory multiprocessor,” in *Proceedings of the 1989 International Conference on Parallel Processing*, vol. 3, pp. 242–249, August 1989.

[7] R. M. Fujimoto and M. Hybinette, “Computing global virtual time in shared-memory multiprocessors,” *ACM Trans. Model. Comput. Simul.*, vol. 7, no. 4, pp. 425–446, 1997.

[8] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. "John Wiley and Sons, Inc.", 2000.

[9] K. Garachorloo et al. “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors”, *Proceedings of the 17th Annual Symposium on Computer Architecture*, pp. 15–26, 1988.

[10] Intel, “Intel Itanium-II Reference Manuals”, Available via the web at: `http://www.intel.com/design/itanium/documentation.htm?iid=ipp_srvr_proc_itanium2+techdocs&`

[11] Intel, “Intel Pentium 4 and Xeon Processor Optimization Reference Manual,” Available via the web at: `http://developer.intel.com/design/pentium4/manuals/248966.htm`

[12] D. Jefferson, “Virtual Time”, *ACM Trans. Prog. Lang. and Systems*, vol. 7, no. 3, July, pp. 404–425, 1985.

[13] D. Jefferson, “Virtual Time II: Storage Management in Distributed Simulation”, In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, August, pp. 75–89, 1990.

[14] L. Lamport, “How to Make a Multiprocessor Compute that Correctly Executes Multiprocess Programs”, *IEEE Transactions on Computers*, vol. 28, no. 9, September, pp. 690–691, 1979.

[15] J. Laudon and D. Lenoski, “The sgi origin: a ccnuma highly scalable server,” pp. 241–251, 1997.

[16] Y-B. Lin and E. D. Lazowska, “Determining the Global Virtual Time in a Distributed Simulation”, In *Proceedings of the 1990 International Conference on Parallel Processing*, vol. 3, August, pp. 201–209, 1990.

[17] Y-B. Lin, “Determining the Global Progress of Parallel Simulation with FIFO Communication Property”, *Information Processing Letters*, 50, pp. 13–17, 1994.

[18] F. Mattern. “Efficient Distributed Snapshots and Global virtual Time Algorithms for Non-FIFO Systems”, *Journal of Parallel and Distributed Computing (JPDC)*, vol. 18, no. 4, August, pages 423–434, 1993.

[19] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. N. Mudge, and R. B. Brown, “Design tradeoffs for software-managed TLBs,” in *ISCA*, pp. 27–38, 1993.

[20] R. Ostrovsky and B. Patt-Shamir, “Optimal and efficient clock synchronization under drifting clocks” In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pp. 3 - 12, 1999.

[21] C. M. Pancerella, and P. F. Reynolds, “Disseminating Critical Target-Specific Synchronization Information in Parallel Discrete-Event Simulation”, In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS ’93)*, pp. 52–59, 1993.

[22] B. R. Preiss, “The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environments”, In *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 21, March, pp. 139–144, 1989.

[23] A. Tomlinson and V. K. Gang, “An Algorithm for Minimally Latent Global Virtual Time”, In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS ’93)*, pp. 35–42, 1993.

[24] B. Samadi. “Distributed Simulation, Algorithms and Performance Analysis”, Ph. D. Thesis, University of California, Los Angeles (UCLA), 1985.

[25] G. Yaun, C. D. Carothers, and S. Kalyanaraman, “Large-scale TCP models using optimistic parallel simulation,” In *Proceedings of Workshop on Parallel and Distributed Simulation (PADS 2003)*, June 2003.

[26] Z. Xiao, J. Cleary, F. Gomes, and B. Unger. “A Fast Asynchronous Continuous GVT Algorithm for Shared Memory Multiprocessors Architectures”, In *9th Workshop on Parallel and Distributed Simulation (PADS ’95)*, June, 1995.

[27] S. Zhou, W. Cai, S. J. Turner and F. Lee, “Critical Causality in Distributed Virtual Environments”, In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS ’02)*, pp. 53–59, 2002.