ANALYSIS OF TIME WARP ON A 32,768 PROCESSOR IBM BLUE GENE/L SUPERCOMPUTER

Akintayo O. Holder[(a)] **and Christopher D. Carothers**[(b)]

[(a)(b)] Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, U.S.A.

[(a)]holdea@cs.rpi.edu, [(b)]chrisc@cs.rpi.edu

ABSTRACT

The aim of our work is to investigate the performance and overall scalability of an optimistic discrete-event simulator on a Blue Gene/L supercomputer. We find that strong scaling out to 16,384 processors is possible. In terms of event-rate, we observed 853 million events per second on 16,384 processors for the PHOLD benchmark. This is 1.5 times faster than any previously reported PDES synchronization protocol for PHOLD executing on a Blue Gene/L supercomputer (e.g. conservative, optimistic or hybrid). Additionally, we observed 2.47 billion events per second for a PCS telephone network model when executed on 32,768 processors. To the best of our knowledge, this is the first multi-billion event rate achieved for any Time Warp model executing on a Blue Gene/L supercomputer.

Keywords: simulation, high performance computing, parallel discrete event simulation

1. INTRODUCTION

Time Warp (Jefferson 1985) is a synchronization protocol that allows a parallel discrete-event simulation to speculatively process event computations, but if the synchronization mechanism detects an event that has been processed out of time stamp order (e.g. event causality error) it will undo or *roll back* the offending event computations. The most common technique for realizing rollback is *state-saving*. Here, the original value of the state is saved prior to event execution. Upon rolling back, the state is restored by copying back the stored value.

In this paper we make two contributions; we demonstrate that is possible to construct an efficient Time Warp simulator which achieves linear scalability on the Blue Gene/L supercomputer out to 16,384 processors and continues to increase its event rate out to 32,768 processors. We also demonstrate that a synchronous Global Virtual Time (GVT) algorithm, which defines a lower bound on any unprocessed or partially processed event, can scale to 10's of thousands of processors.

Rensselaer's Optimistic Simulation System (ROSS) is the basis for this experimental performance study. ROSS was originally a Time Warp simulation engine that was optimized for shared memory systems and is based on Georgia Tech Time Warp (GTW) (Carothers, Perumalla and Fujimoto 1999). This version, which we now call *ROSS-SM*, has some key features that are important for efficient Time Warp execution. First, ROSS-SM efficiently manages memory consumption (Carothers, Perumalla and Fujimoto 1999) for both forward as well as rolled back event computations. In particular, pointers to events are used whenever possible rather than creating duplicate copies. This feature eliminates the need for searches when performing event cancellation (e.g. direct cancellation (Fujimoto 1989)). Using reverse computation also reduces the amount of state saved and has been shown to dramatically improve parallel performance (Carothers, Bauer and Pearce 2000). Finally, GVT is computed using Fujimoto's algorithm (Fujimoto and Hybinette 1997), an asynchronous algorithm that uses a shared global flag to signal GVT computation.

The Blue Gene/L is a completely different class of parallel system. Here, modest computing nodes are connected by a low latency, high bandwidth interconnects, including an independent collective network. Broadcast latency is comparable to point to point messaging, making global reduction efficient. The Blue Gene/L does not allow processors to directly access remote memory, but it includes an efficient message passing framework using MPI (MPI 1994). Consequently, our challenge is to migrate our efficient shared-memory implementation into a highly optimized message passing system that is capable of scaling to supercomputing class processor counts.

The following issues are addressed as part of our im-

plementation of ROSS on the Blue Gene/L which we call *ROSS-MPI*: (i) the sharing of events between processes, (ii) the impact of remote communication on memory consumption, (iii) the role of global virtual time computation on fossil collection, (iv) identifying unique events and ensuring stable, deterministic execution.

The remainder of this study is organized as follows: a brief overview of the IBM Blue Gene/L supercomputer is provided in Section 2. We then discuss the details of the ROSS-MPI implementation in Section 3 and present our synchronous GVT algorithm in Section 3.3. The performance results are presented in Section 4. Finally, related work is discussed in Section 5 with closing remarks in Section 6.

2. BLUE GENE/L SUPERCOMPUTER

The Blue Gene/L is an ultra large-scale supercomputer system that is capable of having 131,072 processors. The Blue Gene philosophy is that more powerful processors is not the answer when it comes to winning the massively parallel scaling war (Adiga et al. 2002). Instead, the Blue Gene architecture balances the computing power of the processor against the data delivery speed of the network. This led designers to create smaller, lower power compute nodes (only 27.5 KW per 1024 nodes) consisting of two IBM 32-bit PowerPCs running at only 700 MHz with a peak memory per node of 1 GB. A rack of Blue Gene is composed into 1024 nodes consisting of 32 drawers of 32 nodes in each draw. Additionally, there are specialized I/O nodes that perform all file I/O. Nominally there is one I/O node for every 32 compute nodes.

Interconnecting both drawers of nodes and racks are five specialized primary networks. The first is the point-to-point network which allows data to be sent between nodes. This network is a 3-D torus consisting of 12 directional links with a bandwidth of 175 MB/s each in the $+-x$, $+-y$ and $+-z$ directions. The latency of a point-to-point message is a function of the distance traveled between nodes. The 32,768 processor Blue Gene/L used in this study consists of 16 racks with each rack being a 32x32x1 torus yielding a network of 32x32x16. The max distance is the sum of half the distance for each direction which is 40 (e.g., $16+16+8$) leading to a max delay of 4 μs (i.e., each hop has a max delay of 100 ns).

In addition to the point-to-point network, there is a global collective network that enables data collection, reduction and redistribution to all nodes (or a subset) with a latency of 5 μs. As we will see in Section 3.3, this collective network is critical to Time Warp's ability to efficiently compute Global Virtual Time (GVT) and re-claim memory. We observe here that the collective network is able to compute a global reduction operation across *all processors* almost as fast as the single longest 1-way delay of the point-to-point network. This suggest that any GVT algorithm using the point-to-point network will not scale as well as using the collective network.

Next, there is an independent barrier network that is able to complete a barrier of a full 64K node Blue Gene/L system in less than 1.5 μs. Finally, there is a separate control network used to transmit system health information as well as an Gigabit Ethernet network which provides connectivity between I/O nodes and an external parallel file system.

For this experimental study, the Blue Gene/L housed within the Rensselaer Computational Center for Nanotechnology Innovations (CCNI) is used. This is a 16 rack Blue Gene/L system with 8 racks having 512 MB of RAM per node and the other 8 racks configured with 1 GB of RAM per node. The IBM XLC C compiler was used for all the results in this paper. We were able to take full advantage of the compiler's peak optimization level as well as architecture specific settings. Our specific compiler options where: `-O5 -qarch=440d -qtune=440`.

3. ROSS-MPI IMPLEMENTATION

In the ROSS implementation of the Time Warp protocol, the processor element (PE) is an abstraction of the physical processor which we realize as an MPI task. They are independent processes that have exclusive memory access and communicate via message passing. Events that are destined for a logical process (LP) on another PE are sent as MPI messages to the correct task. Each PE owns a number of LPs and uses a master scheduler to process events in time stamp order for all LPs assigned to that PE. Under the Time Warp protocol, models are implemented as parallel applications where an LP would be a logical thread of execution. The event handler is executed by the event's destination LP, as the LPs are responsible for processing and scheduling events. The models are built from LPs and events, while the PEs and KPs are architectural features used by ROSS. This allows the model to define parallelism independently of the processor count. The model provides the addressing protocol for routing events among LPs and PEs. Next, because each physical processor has it own memory, remote events are duplicated at the source and destination PEs. This duplicaton allows us to implement a rollback that spans across physical processors and separate memory address spaces. Remote events, and how they are handled will be important to the performance of ROSS-MPI. Last, a synchronous GVT algorithm was implemented that exploits the Blue Gene/L's collective network to achieve scalable performance.

Kernel processes (KPs) (Carothers, Bauer and Pearce 2000) improve the efficiency of fossil collection by aggregating events processed for a group of LPs into a larger list. ROSS-MPI maps the LPs to the KPs and the KPs to the PEs, but it is the responsibility of the model to assign LPs their identifiers. When a PE needs to schedule an event

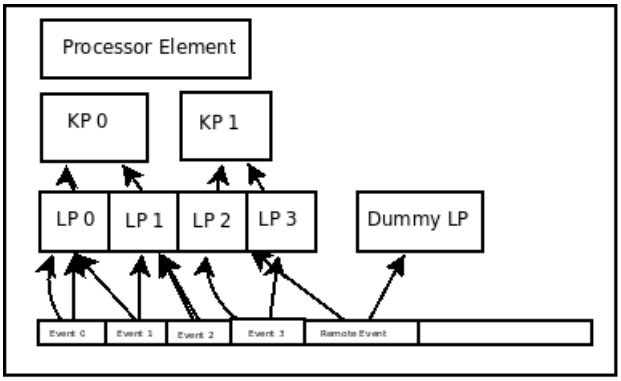

Figure 1: Architecture of a Processor Element (PE)

on another PE, it uses the model's LP addressing scheme, which will be described below. Remote events introduce the need to differentiate among events, so we use `age` a sequence number that uniquely identifies all the events generated between any pair of LPs. Because the MPI standard (MPI 1994) guarantees in order delivery of messages, event arrival order is preserved thus avoiding the complex case of having to process cancel events prior to the receipt of their positive event counterpart. This aspect is described in more detail in Section 3.2.1. Finally, the GVT algorithm empties the receive buffers and uses collective operations to ensure all processor elements agree that there are no transient events. We define an overflow buffer to be used for receiving remote events during GVT computation. This overflow buffer allows GVT computation to proceed when regular event memory has been exhausted.

3.1. Processor Element Communication

As shown in Figure 1, each MPI task contains a PE data structure, arrays of LPs and KPs, and a free list of events. The free list is an allocation of all the events that will be used over the life of the simulation. ROSS does not allocate memory during allocation, rather it provides a reference to an event in the free list. These are distributed equally among processor elements which should ensure that each processor performs an equal share of the work. ROSS-MPI assumes that the model will be well balanced, but better support for unbalanced models will be available in the future. This would include a more general LP to KP mapping structure, that will allow variance in the number of LPs assigned to a KP. We do believe that a balanced model is best suited for large platforms, like the Blue Gene. However, we acknowledge the need to support models where LPs are assigned varying workloads.

ROSS-MPI maps the LPs to KPs, but models may take advantage of the scheme when defining their LP addressing scheme. ROSS-MPI ensures that the i^{th} KP is assigned the $((i-1)(nlp/nkp)+1)$, $((i-1)(nlp/nkp)+2)..(i)(nlp/nkp)$ LPs, where nlp and nkp are the number of LPs and KPs respectively. Each LP is assigned an address that is comprised of an `index` and a `peid`, the model then assigns an `lpid`. The `peid` refers to the processor element that contains the LP, and the `index` is the location of the LP in the local array of LPs. The `lpid` is a unique identifier that is used by the model. The `map_lp_to_pe` function maps the `lpid` to the `peid` and the `map_lp_to_local` function maps it to the `index`. We chose to always compute the address pair consisting of the `index` and the `peid`, as opposed to constructing a table that describes the mapping or caching the computed results. This is a space-computation trade-off. A mapping table would limit the scale of the model that could be executed since mappings for all the LPs would be retained on each MPI task and caching would increase complexity of the implementation. Based on our current performance results, we have seen no indication that the use of efficient mapping functions degrades performance.

3.2. Remote Events

Remote events are generated when the source and destination LPs are not on the same PE. The source LP creates the remote event and places it on the current event's `caused_by_me` list. The "current event" is the event that is being processed by the LP during event generation. Next, the source LP sends the event to the PE that hosts the destination LP. When the destination PE receives an event it finds the correct destination LP, and inserts the event in the priority queue. If the source LP needs to rollback, it will send a remote cancel event that contains a duplicate of the remote event. ROSS only uses the source LP, destination LP, time stamp and age to identify the correct event, but a complete duplicate is sent. The mapping functions are used by the PEs to find the source and destination LPs of an event.

When a PE sends a remote event, it uses `map_lp_to_pe` to compute the destination `peid`. The destination PE, upon receiving a remote event, uses the `map_lp_to_index` to find the local LP that corresponds to the destination `lpid`. These mapping functions must initialized before the scheduler loop begins as any LP could be referenced once event processing commences. The model builder must ensure that all the `peid`, `index` pairs are mapped to `lpids`.

3.2.1. Augmenting Direct Cancellation

When a remote event or cancel arrives, the PE must be able to tell if the event is unique or a copy of an existing event. A remote cancel without the associated event is an error, as is the duplication of a remote event. With ROSS-SM every event is made unique by leveraging shared memory to perform direct cancellation (Fujimoto 1989). Here, a "cancel" message/event is a pointer reference to the actual real

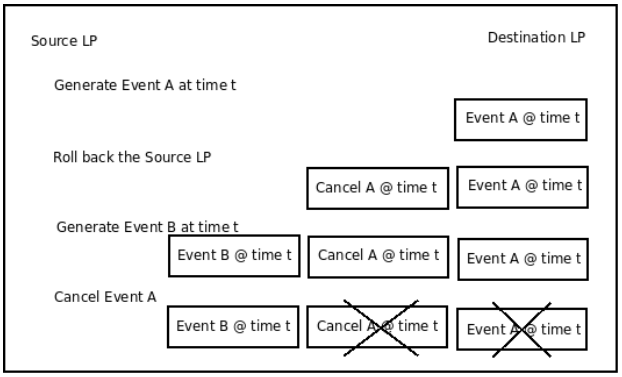

Figure 2: Handling concurrent events due to roll back

event, thus making the positive event and the anti-message the same object. With ROSS-MPI, the PE must search the priority queue and processed list to cancel a previously sent remote event. The use of KPs (Carothers, Bauer and Pearce 2000) allows us to perform a quick linear search of the processed list. The search of the priority queue, which is a Splay Tree data structure, is derived from the insert method which has a complexity of $O(log(n))$ (Sedgewick 1998). To implement a search we must be able to differentiate among the events in the simulation. Consider the case shown in Figure 2. Here, a source LP schedules an event A at time t to a destination LP. The source LP is then rolled back and it generates an anti-message for event A. After rollback is complete, the source LP schedules a new event B at time t on destination LP. When processing the cancel for A we must be aware that the new event B could have the same time stamp despite being the "correct" event. Thus, we need a mechanism to uniquely identify events that are to be cancelled. To accomplish this, we identify an event using these four fields: (i) the event's simulated time at which it is to be processed `recv_ts`, (ii) source lp `src_lpid`, (iii) destination lp `dest_lpid`, and (iv) a sequence number `age`. Every logical process maintains an non-decreasing sequence counter, when it generates an event it assigns the event the current value as its `age`, and then increments the counter. This `age` will be different for each event generated by a given LP. In the scenario we discussed, event A and cancel A would have the same age while B, the newer event, would be different. This happens because cancels are copies of the original event, and only new events are given unique `ages`.

3.2.2. Correctness and efficiency

A remote cancel event is a copy of the original event that is sent with a different MPI tag. Using a stub event, like DSIM (Chen and Szymanski 2007), while reducing network traffic, appears to complicate event handling and may not improve memory efficiency. At the source PE, we fossil collect the remote event when their parent is being collected. We exploit the observation that only the parent has a reference to the remote event. When fossil collecting, we check the `caused_by_me` list and fossil collect all remote events. This approach allows us to reclaim memory efficiently and eliminates the need to manage lists of remote events.

ROSS-MPI retrieves remote events by combining blocking receives with non-blocking probes. The non-blocking probe signals when a remote event is available, but it will not block if the buffers are empty. The combination gives us the semantics of non-blocking receives with simpler code and without the use of expensive MPI operations like message cancel. We use the `MPI_ANY_TAG` to poll both event types, rather than retrieve events and cancels separately. This solves the following race condition: if the event and cancel arrive when the PE is retrieving cancels, the cancel, which is processed first, will appear to be unaccompanied. By polling both ports and checking the event type after retrieval the problem is avoided. This solution is dependent on the semantics of MPI point-to-point communication which guarantees that "messages are non overtaking" (MPI 1994). The race condition is avoided since an event/cancel pair is sent between the same two LPs, and a cancel can only be generated after the initial event.

3.3. GVT algorithm

A consistent cut (Mattern 1994) divides the events into past and future. Here, no events would be sent from the future into the past. If all the processors agree to the cut, the Global Virtual Time (GVT) is the time stamp of the earliest event in the present.

Algorithm 1 is a global reduction (Chen and Szymanski 2007) GVT algorithm, it is synchronous and it uses collective operations to create a consistent cut. When the processors reach the synchronization point they ensure that all transient messages are accounted for by performing a collective sum over the count of outstanding messages. Once all messages are accounted for, the cut is formed and then the processors perform a collective minimum over their local virtual time (LVT) which consists of the minimum of any event in the priority queue or otherwise awaiting processing. The latency of broadcast on the Blue Gene/L makes a synchronous global reduction GVT algorithm an efficient choice.

3.3.1. Correctness

A correct GVT algorithm must solve the transient message and simultaneous reporting problems (Fujimoto and Hybinette 1997). We present arguments that show our algorithm addresses both problems. A transient message is a message that is not visible to any processor because it is traversing the network. The simultaneous reporting problem exists where both the sender and receiver expect the other to ac-

Algorithm 1 GVT Computation algorithm

Require: global_message_counter, the difference between events sent and received since last GVT.
Require: local gvt estimate, earliest remote event since the last GVT.
Ensure: GVT is the minimum of all unprocessed events

```
message_counter = 0
repeat
  while incoming messages available do
    read(remote event)
    decrement global_message_counter
    if local gvt estimate > remote event time stamp
    then
      local gvt estimate = remote event time stamp
    end if
    enqueue remote event
  end while
  MPI_Allreduce(global_message_counter,
  message_counter, SUM)
until message_counter == 0
global_message_counter = 0
LVT = min(earliest event in priority queue,
earliest unprocessed cancel event)
MPI_ReduceAll(LVT,GVT,MIN)
```

count for the event in their respective notions of local virtual time (LVT).

Transient message, by induction. Basis: Consider the base case at time 0, no remote events have been created, so the sum of the differences between the messages sent and received is zero. There are no transient messages.

Inductive step: For a given epoch, we increase the counter for each message and decrease for each reception. At the end of the epoch the counter is zero. If a transient message exists, it would be sent during or before the current epoch. If the transient message was sent during this epoch, the counter must be negative when we entered the epoch. If the transient message was sent before, the counter must be positive when we entered. Since the counter is always zero when an epoch ends neither is possible. □

Simultaneous reporting, by contradiction. Assume the simultaneous reporting problem exists. This implies that a PE must have received the GVT computation request after it has transmitted the earliest event in the system and the destination PE already responded the GVT request and sent its LVT before it has received the earliest event. In this case, each PE believes the other responsible for the inclusion of this earliest event in there LVT calculation. So GVT is being computed while the event is being transmitted. Our prior argument shows, there are no messages in-flight when GVT computation commences. □

3.4. Memory Management

Memory management in ROSS-MPI focuses on the impact of remote events. ROSS-SM is a memory efficient Time Warp implementation but remote events have the potential to double memory consumption, as previously noted in Section 3.2. The pathological cases will always cause problems, but we take the following steps to manage memory consumption. First, overflow buffers are allocated for use when receiving events during GVT computation and, at the source PE we fossil collect remote events with their parents.

The overflow buffer is a one time allocation specified by the model, and is only used when the PE's free list has been exhausted. Events allocated from the overflow buffer are added to the PE's free list when deallocated, they are not returned to the overflow buffer. The number of events allocated to the overflow buffer is defined by `tw_events_gvt_compute`. The overflow buffer is only used for retrieving remote events during GVT computation, and its length should be dependent on the communication pattern of the model. It should be proportional to number of iterations through the full ROSS-MPI scheduler loops which is set by `g_tw_gvt_interval`, and the size of the processing batches ,`g_tw_mblock`. If the PE's free list is empty, it will abort the scheduler loop and begin to compute GVT, and then start fossil collection.

3.5. Eliminating Global Data Structures and Non-Deterministic Event Processing

In addition to efficiency and correctness, we ensure the simulator scales well by eliminating global data structures and globally redundant operations. Instead of a global array of random number generators, we distributed each random number generator as part of the LP data structure. The stream is reversible (Carothers, Perumalla and Fujimoto 1999), allowing the reverse computation to "undo" any previous calls to the random number generator. The new fields in the LP data structure are the arrays that contain the seeds of the random stream and the variables used in sampling other distributions. The streams are seeded sequentially, so given the `lpid` we can calculate how many seeds to skip and initialize the stream with limited redundant work. A nice side effect of this data structure design change is that it co-locates the random number seeds with the LP state which enables better LP data locality for improved cache performance during event processing.

We also address the increased probability that two events may have the same destination LP and time stamp. As discussed in (Wieland 1997) simulations with running times that are large compared with the granularity at which events are scheduled increase the probability that two or more events will have the same destination LP and time stamp, if not the same source LP. To ensure deterministic event processing

in the face of event simultaneity, the source LP, destination LP and the age of an event will be considered when sorting events in priority queue. This allows every event to be uniquely identified, and ensures the ordering is independent of the insertion order. We also rollback when the new event is equal to or older than the last processed event. This insures that all events with the same time stamp will be executed in deterministic order, but at the expense of increased rollbacks. However, because this behaviour occurs infrequently in real applications, we do not believe overall performance is degraded. By rolling back under these conditions, we must consider the possibility of livelock if there is a cycle of LPs scheduling events with zero delay. We do not believe this pathological case to be a reasonable situation in a model.

4. EXPERIMENTAL RESULTS

PHOLD is a synthetic benchmark, commonly used for testing the performance of Time Warp simulators (Chen and Szymanski 2005) and (Perumalla 2006). PHOLD has minimal event processing, minimal look ahead due to event scheduling being based on a random distribution and a random communication pattern. PHOLD can be configured by changing the event population and the ratio of remote events. PHOLD was configured to schedule 10 percent of events to remote LPs. The simulations had 1024x1024 or 1,048,576 LPs and an initial event population of either 10 or 16 events per LP. The 10 events per LP case and 10 percent remote ratio are comparable with the PHOLD configuration used by Permulla's performance study in (Perumalla 2007).

The second workload model is a PCS network that provides wireless communication services for cellular phone subscribers. Here, the service area of a PCS network is populated with a set of geographically distributed transmitters/receivers called *radio ports*. A set of radio channels are assigned to each radio port, and the users in the *coverage area* (or *cell* for the radio port) can send and receive phone calls by using these radio channels. When a user moves from one cell to another during a phone call a *hand-off* is said to occur. In this case the PCS network attempts to allocate a radio channel in the new cell to allow the phone call connection to continue. If all channels in the new cell are busy, then the phone call is forced to terminate. It is important to engineer the system so that the likelihood of force termination is very low (e.g., less than 1%). What is special about this application is that it is an instance of a class of applications that are *self-initiated* (Nicol 1991). Here, LPs typically schedule most of their events to themselves, which leads to fewer remote messages relative to locally scheduled events making this class of applications well suited for ultra large processor count systems like the Blue Gene/L. For a detailed explanation of our PCS model, we refer the reader to (Carothers, Fujimoto and Lin 1995).

The PCS model had a grid of 4096x4096 PCS cells which results in LPs with a constant configuration except for the simulated time, which was doubled for the 32,768 processor run.

4.1. PHOLD Performance

Because our access to the Blue Gene/L is limited, we where only able to perform one run at each processor count configuration. However, as well will demonstrate, the Blue Gene/L's performance has very little variation unlike typical multi-user clusters and thus we believe these single run performance numbers to be very close to the multi-run averages. Additionally, the CCNI Blue Gene/L partitions are predefined with the following node counts: 512, 1024, 4096, 8192 and 16384. Thus, in order to maxmially utlize the processor counts in each available partition class, we made runs using processor counts of 1024 using a 512 nodes, 2048 using 1024 nodes, 8192 using 4096 nodes, 16,384 using 8192 nodes and 32,768 using 16,384 nodes.

Unlike typical large-scale clusters, the Blue Gene/L has no virtual memory and does not support a multi-program environment where OS-level daemons co-exist and compete for resources with user-level compute jobs. Thus, we found very little variance in performance across multiple runs. In order to demonstrate repeatable performance, we used multiple runs on smaller processor partitions. Figure 3 shows the event rate of PHOLD over 10 runs with 2048 processors. The standard deviation of the event rate was 0.05% of the mean, with PCS (not shown) it was 0.01% of the mean.

Figure 4 shows the aggregate event rate of PHOLD as a function of processor count. For the 16 events per LP case, we observed a rate of 43 million events per second on 1024 processors, which increased to a peak event rate of 798 million on 16,384 processors. Additionally, we observed an increase in event rate for the 10 events per LP case. The peak event rate was 853 million. This is about 1.5 times better than the peak rate described by Perumalla (Perumalla 2007), using any PDES synchronization scheme (e.g. conservative, optimistic or hybrid) on the same hardware. We believe the increase in performance for the 10 events per LP case is due to lower priority queue overheads as consequence of the smaller per processor event population.

For the 10 events per LP case, the per processor event rate was 45,000 on 2048 processors and it increased to 52,000 with 16,384 processors. The speedup of PHOLD appears to be super-linear despite the use of strong scaling. A lack of available work was a concern given that the 1,048,576 LPs are spread over 16,384 processors yielding only 64 LPs per processor. We attribute the super-linear performance to a decrease in priority queue overheads as the processor count increases. Recall, we use a Splay Tree data structure for the priority queue which as both a $O(log(n))$ complexity for

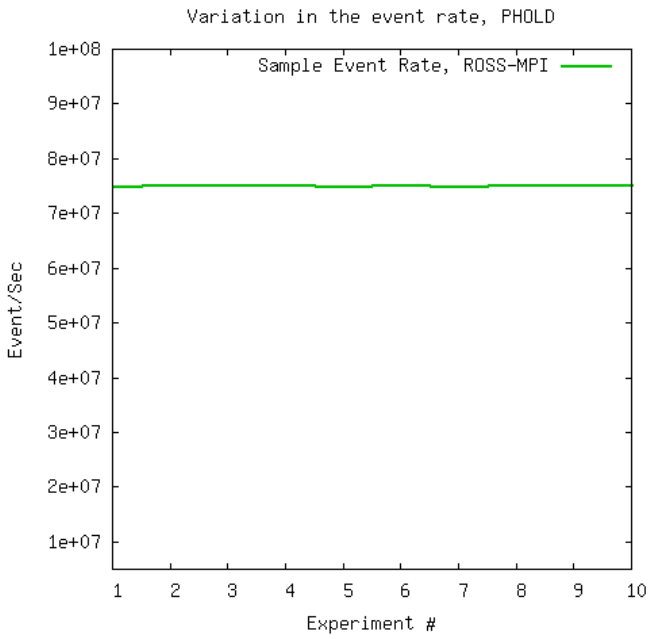

Figure 3: Event Rate for PHOLD across multiple runs on 2048 processors for 16 events per LP .

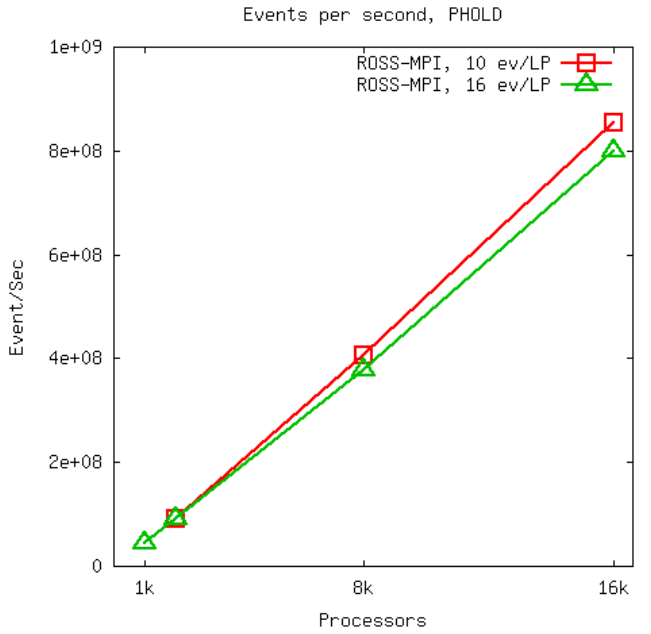

Figure 4: Event rate for PHOLD for both 10 events per LP and 16 events per LP as a function of processor count.

enqueue and dequeue operations. As the processor counts double, the event population per processor goes down by half, which decreases priority operations by a log(2) factor.

Figure 5 shows the number of rollbacks, primary and secondary for the 16 events per LP case as a percentage of committed events. This rollback ratio increased, almost

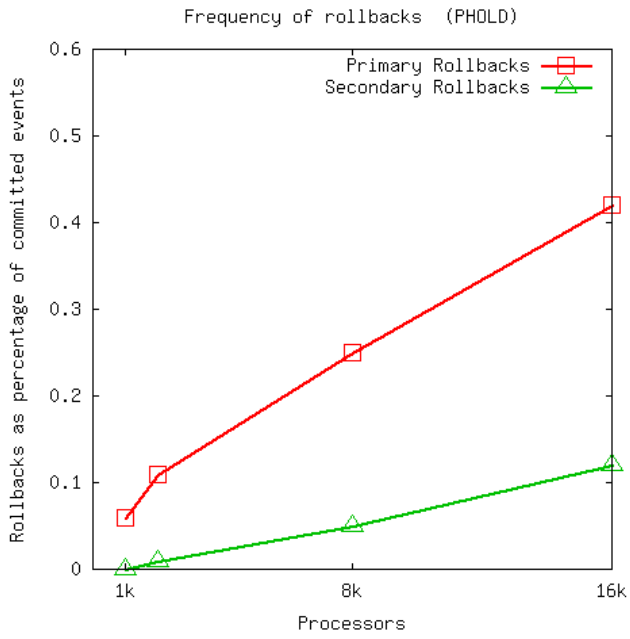

Figure 5: Primary and secondary rollbacks for PHOLD for 16 events per LP case as a function of processor count.

linearly, with the increase in processor count. Both primary and secondary rollback ratios appeared to be gradually increasing, but we only observed a total rollback ratio of 0.54% with 16,384 processors. These low rollback ratios indicate the processors spend most of their time doing useful work.

4.2. PCS Performance

Figure 6 shows that our event rate increases almost linearly to 2 billion events per second on 16,384 process, but beyond that the increase is sub-linear. The 25% increase in performance implies that the per processor event rate fell by 40% as we increased to 32,768 processors. The peak event processing rate was 2.47 billion events per second on 32,768 processors. Figure 7 shows that the ratio of rollbacks is less than 0.06% of the committed events. We do not believe changes in the ratios are significant as they remain low. The performance decrease could due to the size of the model, or limitations in the Blue Gene/L hardware, especially the collective network. The longer run for the 32K processor case was not believed to be responsible for the sharp decrease in per processor event rate. Further investigation is needed before we fully understand this phenomenon.

5. RELATED WORK

There have been some investigation into the performance of discrete event simulation on supercomputers with more than 1000 processors. The performance of conservative,

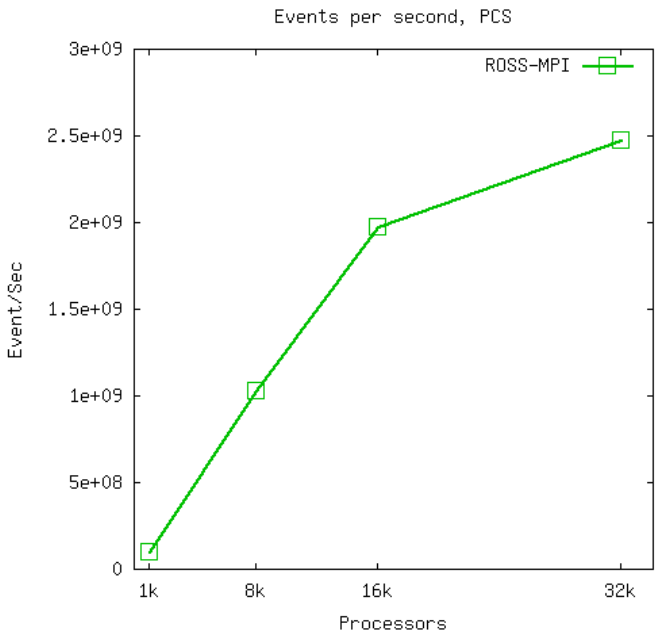

Figure 6: Event rate for PCS Model as a function of processor count.

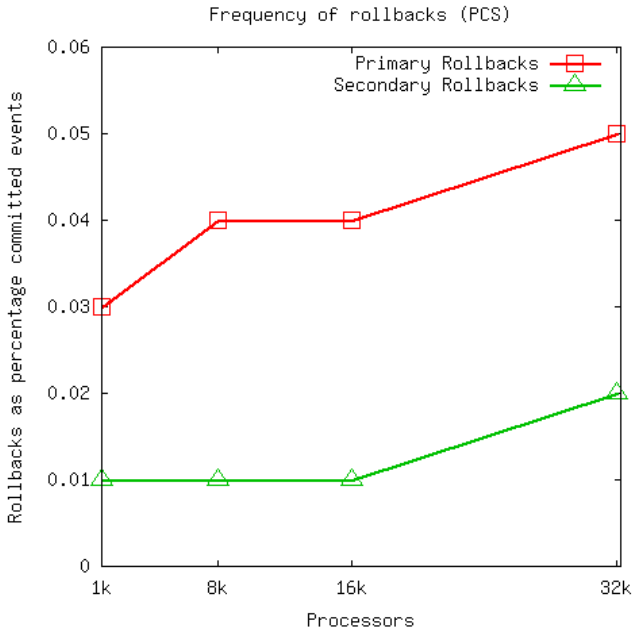

Figure 7: Primary and secondary rollbacks for PCS Model as a function of processor count.

optimistic and other approaches to PDES on the Blue Gene/L has been examined (Perumalla 2007). The observed performance may have been limited by the use of strong scaling and limited access to the Blue Gene/L. In a related Blue Gene Consortium report (Perumalla 2007), they describe porting the SCATTER road network model, highlighting the issue of porting existing models. PHOLD and PCS are well balanced models, we consider load balancing to be beyond the scope of this paper, but it would be interesting to consider problems where this is an issue.

The performance of PDES on a 750 node Alpha server has also been investigated (Chen and Szymanski 2005). They were able to process an impressive 228 million events per second on 1024 processors. Their PHOLD model schedules events at one of the four nearest neighbors, which should exploit the quad processor SMP nodes while avoiding the Quadrics network switch. DSIM uses, Time Quantum GVT (Chen and Szymanski 2007), a manager-worker GVT algorithm which reserves processors as GVT managers. They estimate that one manager is needed for every 128 processors, but the approach appears to be scalable.

In the context of GVT algorithms based on hardware-based acceleration approaches, there been some activity in the past. Of note, Pancerella and Reynolds (Pancerella and Reynolds 1993), they present results of simulations that suggest that hardware assisted, target-specific global reductions can dramatically improve parallel simulator performance. More recently Noronha and Abu-Ghazaleh (Noronha and Abu-Ghazaleh 2002) has shown the benefits of offloading the GVT computation to network interface cards. However, here Mattern's asynchronous GVT algorithm (Mattern 1994) is used as opposed to a synchronous reduction network-based algorithm.

6. CONCLUSIONS

We demonstrate that it is possible to construct an efficient optimistic Time Warp simulator which achieves linear scalability on the Blue Gene/L supercomputer, and we were able to process almost 2 billion events per second on a 16,384 processors for the PCS model. This would indicate that the optimistic approach to PDES has strong scaling potential on the largest scale supercomputers of today. It also demonstrates that a synchronous GVT computation algorithm can be used in an efficient Time Warp implementation provided that the underlying supercomputer architecture supports high-performance global collective operations. These results suggest that a synchronous algorithm, which can exploit the presence of a global collective/reduction network, will outperform a point-to-point solution, especially when scaling to near-petascale supercomputer system.

As future work, we plan to compare how the Blue Gene performs relative to other supercomputer architectures such as Texas Ranger, AMD Quad-Core Opteron Cluster and the Cray XT3 MMP system which are both part of the TeraGrid (see `www.teragrid.org`).

REFERENCES

Adiga, N.R and et al., 2002. An overview of the Blue Gene/L Supercomputer. *Proceedings of the ACM/IEEE Conference on Supercomputing*, pp. 1–22, November 16–22 Baltimore, Maryland, USA.

Carothers, C.D., Bauer, D. and Pearce, S., 2000. ROSS: A High-Performance, Low Memory, Modular Time Warp System, *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pp. 53–60, May 28 - 31, Bologna, Italy.

Carothers, C.D., Fujimoto, R.M. and Lin, Y-B. 1995. A case study in simulating PCS networks using Time Warp. *Proceedings of the 9th workshop on Parallel and Distributed Simulation*, pp 87–94, June 13 - 16, Lake Placid, New York, USA.

Carothers, C. D., Perumalla, K.S., and Fujimoto, R.M., 1999. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation* 9(3):224–253.

Chen, G. and Szymanski, B.K., 2005. DSIM: scaling Time Warp to 1,033 processors. *Proceedings of the 37th conference on Winter simulation*, pp 346–355. December 4-7, Orlando, Florida, USA.

Chen, G. and Szymanski, B.K., 2007. Time quantum GVT: A scalable computation of the global virtual time in parallel discrete event simulations. *Scalable Computing: Practice and Experience:Scientific International Journal for Parallel and Distributed Computing*, 8(4):423–436.

Fujimoto, R.M. and Hybinette. M., 1997. Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation*, 7(4):425–446.

Fujimoto, R.M., 1989. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computing Simulation International*, 6(3):211–239.

Jefferson, D.R., 1985. Virtual time. *ACM Transactions on Programming Language Systems*, 7(3):404–425, 1985.

Mattern, F., 1994. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(3):423–434.

Message Passing Interface Forum., 1994. *MPI: A message-passing interface standard*. Message Passing Interface Forum. Available from: http://www.mpi-forum.org/ [accessed 16 March 2008]

Noronha, R. and Abu-Ghazaleh, N.B., 2002. Using programmable nics for Time Warp optimization. *Proceedings of the International Parallel and Distributed Processing Symposium*, April 15-19, Fort Lauderdale, Florida, USA.

Nicol, D.M., 1991. Performance bounds on parallel self-initiating discrete-event simulations. *ACM Transactions on Modeling and Computer Simulation*, 1(1):24–50.

Pancerella, C. and Reynolds, P.F., 1993. Disseminating critical target-specific synchronization information in parallel discrete event simulation. *: Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pp 52–59, May 16 - 19, San Diego, California, USA.

Perumalla, K.S., 2006. *Ultra-scale parallel discrete event applications*. Oak Ridge National Laboratory. Available from: `http://www.bgconsortium.org/` [accessed 16 March 2008]

Perumalla, K.S., 2007. Scaling Time Warp-based discrete event execution to 10**4 processors on a Blue Gene supercomputer. *Proceedings of the 4th international conference on Computing frontiers*, pp 69–76, May 7-9, Ischia, Italy.

Sedgewick, R., 1998. *Algorithms in C*. 3rd ed. Boston:Addison-Wesley

Wieland, F., 1997. The Threshold of Event Simultaneity. *Proceedings of the Workshop on Parallel and Distributed Simulation*, pp 56–59, June 10 - 13, Lockenhaus, Austria.

AUTHORS BIOGRAPHY

AKINTAYO HOLDER is a Ph.D. student in the Computer Science Department at Rensselaer Polytechnic Institute. His research interests are parallel and distributed computing.

CHRISTOPHER D. CAROTHERS is an Associate Professor in the Computer Science Department at Rensselaer Polytechnic Institute. He received the Ph.D., M.S., and B.S. in Computer Science from Georgia Institute of Technology in 1997, 1996, and 1991, respectively. His research interests include parallel and distributed systems, simulation, networking, and computer architecture.