DSIM: Scaling Time Warp to 1,033 Processors

Gilbert Chen and Boleslaw K. Szymanski
Department of Computer Science,
Rensselaer Polytechnic Institute
110 8th Street, Troy, NY 12180, U.S.A.
{cheng3,szymansk}@cs.rpi.edu

Abstract

This paper describes the design and implementation of a new Time Warp simulator, called DSIM that targets clusters comprised of thousands of processors. DSIM employs a novel technique for efficient, scalable GVT computation, called the Time Quantum GVT algorithm that requires no message acknowledgement and relies on constant-length messages. In addition, DSIM implements a Local Fossil Collection mechanism that alleviates the overhead associated with memory reclamation and an efficient event management to ensure fast event manipulation during simulation. It is also equipped with a simple programming interface to ease programming and debugging of simulations. Experimental results obtained on the PHOLD benchmark demonstrate that DSIM can process as many as 228 million events per second on 1033 processors.

1. Introduction

Clusters have steadily and gradually become the main stream of high performance computing platforms – the numbers of clusters among the top 500 supercomputer list (http://www.top500.org/) jumped from 208 in November 2003 to 291 in June 2004, an increase of 40% in merely 7 months. It is unlikely that this trend will not continue in the future, as technological progresses on single processors and fast interconnection networks made clusters much easier and more cost-effective to build from commodity components than ever before.

The popularity of clusters brings a new challenge to PDES (Parallel Discrete Event Simulation) research. Historically, at different stages of development, parallel discrete event simulators have always targeted the most widely available platforms at the time. For instance, the first generation Time Warp simulators [1-3] were usually designed for parallel computers, especially MPP (Massively Parallel Processing) machines. When shared-memory computers become more prevailing in the nineties, the second generation Time Warp simulators, such as GTW [4] and ROSS [5], were developed to support this type of parallel computers. Today, as clusters are starting to dominate high-end computing, new Time Warp simulators capable of running efficiently on networks of distributed processors are particularly desired.

There are some Time Warp simulators such as WARPED [6] and ParaSol [7] that were initially designed for distributed memory parallel computers. One may argue that these simulators can be portable to clusters with a little effort, as modern clusters may exhibit behavior that is close or comparable to distributed memory MPP computers, in aspects such as bandwidth, latency, and scale. However, little data have been available to prove these simulators' effectiveness on thousands of processors. In fact, the largest Time Warp simulations that have ever been attempted, in terms of the number of processors used, to the best of our knowledge, were executed on 64 processors [8, 9]. Hence, it is natural to wonder whether Time Warp simulators running on large-scale clusters can attain satisfactorily good performance.

DSIM was developed to support efficient Time Warp simulation on distributed clusters with thousands of processors. At the heart of DSIM is a new GVT (Global Virtual Time) algorithm, referred to as the Time Quantum GVT (TQ-GVT) algorithm that does not require message acknowledgement, and relies on short messages with constant length. This paper demonstrates that TQ-GVT is able to deliver a continuous stream of up-to-date GVT values with minimum delays during simulation. It achieves this by devoting one or more processors to running the core of the GVT algorithm. In view of the large number of processors involved in the simulation on a large cluster, such a solution is more efficient then delaying all processors by a small fraction of time in a distributed implementation of the GVT computation.

In addition to the new GVT algorithm, DSIM uses a modified fossil collection mechanism called Local Fossil Collection, as well as a careful implementation of the event management mechanism. However, DSIM is not just about performance. An equally important aspect of DSIM design is to demonstrate that it is possible to improve the programmability of a parallel discrete event simulator without sacrificing its efficiency.

The primary goal of this paper is to introduce the basic features of DSIM to make readers aware of the availability of yet another Time Warp simulator. The PHOLD model was simply chosen as a non-trivial modeling problem, but it is yet to be demonstrated if the reported performance will be repeatable for other types of models. Likewise, it is yet to be established if new features of DSIM are indispensable for high performance of Time Warp on large-scale clusters. Still, it is safe to predict that some features, such as the new GVT algorithm, may play a larger role in enabling superior performance than, for example, those that are conceived merely for improving programmability.

The paper is organized as follows. Section 2 introduces the basic ideas behind the Time Quantum GVT algorithm. Section 3 presents DSIM implementation details, including Local Fossil Collection and the memory efficient event management mechanism. Section 4 describes the programming interface of DSIM in detail. Section 5 presents the experimental results of DSIM running on two different yet popular clusters. Section 6 concludes the paper by providing a summary of the basic features of DSIM and a vision of its future development.

2. Time Quantum GVT Algorithm

The GVT is defined as the minimum of local simulated times on all processors and timestamps of all messages in transit. A good GVT algorithm is critical to the overall performance of a Time Warp simulator. A processed event earlier than the GVT will not be subject to rollback under any circumstances, and therefore its associated memory can be released permanently. With more accurate GVT estimate, more obsolete memory can be reclaimed, decreasing the chance for a performance loss resulting from the memory system bottleneck.

Numerous GVT algorithms [10-22] have been proposed for distributed execution of Time Warp. Some of them depend upon message acknowledgement [12, 14, 16] to track those messages that are still in transit when the GVT computation is being conducted. Others [11, 17, 19] utilize Vector Clock or similar structures whose size is proportional to the number of processors. Still others [10, 13, 15, 18] take a distributed and passive approach, which requires the processor in need of memory to send out a special token message that will traverse all other processors. Synchronization-based GVT algorithms [20-22] rely on global reductions to determine whether previously sent messages have all been received.

Instead of implementing one of these GVT algorithms, yet another GVT algorithm, referred to as Time Quantum GVT (TQ-GVT), is contributed to the already rich repository. Its uniqueness relies on allocating a processor, called the GVT master, exclusively to the GVT computation. Its name comes from the fact that the simulated time is divided into a continuous sequence of time quanta with equal width, somewhat similar to the approach used in [22]. A GVT master monitors the numbers of sent and received messages within each time quantum for all simulating processors reporting to it. If the two numbers associated with a given time quantum are equal, then all messages sent during this time quantum must have already been received and they will not be counted when computing the new GVT value. This well-known method of accounting from transient messages can be traced back to Mattern [19].

The use of an exclusive processor for GVT computation may appear an obstacle to scalability, since it is basically a centralized approach. Nevertheless, one or more levels of intermediate GVT masters can be introduced, each of which keeps track of the number of transient messages in each time quantum for a subset of processors. These numbers must then be reported to the root GVT master, which in turn determines whether or not there are still messages in transmit from each time quantum and calculates the GVT accordingly. Empirically, one GVT master can drive as many as 128 simulating processors, so an extra level of intermediate masters could easily increase the number of simulating processors to 16,384 and if this is not sufficient, another level of GVT masters can be added. A very small percentage of processors, 129 out of 16,513, or merely 0.78 percent, will not be directly participating in the simulation. Hence, such a solution is suitable for clusters in which the numbers of processors involved in a computation are large.

An essential feature that makes TQ-GVT scale so well is that it overlaps GVT computation with other tasks. Simulating processors are only engaged in processing events and transmitting and receiving messages. With regards to GVT computation, they just need to send report messages periodically, and receive GVT messages as they come. If GVT messages do not come on time, simulating processors are never delayed or blocked, as in the case of some other algorithms. As a result, only the masters are involved in a significant computation of the GVT; the cost of GVT computation for simulating processors is non-blocking send of a report message at the end of each quantum and non-blocking receive of the new GVT value if there is a GVT message. Thus, this solution eliminates two problems: (i) the large latency of the interconnection network often found in clusters, and (ii) the asynchrony that causes different processors to reach the synchronization point at vastly different wall-clock times when the number of involved processor exceeds, say, 1000.

On the first glance, the TQ-GVT looks similar to the LBTS algorithm that chops the simulation time into bands or epochs [22]. However, this superficial similarity disappears as soon as one considers the fact that the performance study in [22] was limited to 16 processors. Three essential differences distinguish TQ-GVT from the LBTS algorithm. First, in [22] multiple reductions may be needed within each band for all transient messages sent during the previous band to be received, while in TQ-GVT, each simulating processor is guaranteed to sent only one report message and receive at most one GVT message per time quantum. Second, in [22] during each band every processor must perform both global reduction and event processing. This, unfortunately, increases the latency of handling global reduction messages because, to maximize the event rates, a certain number of events must be consecutively processed before checking the message receive buffer. In contrast, in TQ-GVT, simulating processors can focus on event processing and message exchanges. It is the GVT master that carries out the global reduction, and it can respond to GVT-related messages more quickly than simulating processors, since this is its only job. Third, and perhaps the most important one, TQ-GVT never delays or blocks simulating processors at the end of the time quantum, and hence no temporary disturbances on communication can impact the progress of time quanta. In [22], however, a new band may start only after all transient messages from the previous band have been accounted for.

Two methods can be used for advancing time quanta. The first method depends upon a hardware clock to advance the time quantum at fixed intervals. Clocks on different processors, however, need not be precisely synchronized, since a processor that is too far ahead or behind only causes some messages to be counted in the next time quantum. The other method simply requires that the GVT master broadcast a special message periodically, upon the receipt of which each simulating processor will advance the time quantum. The latter method requires more bandwidth.

The algorithm is briefly described below:

- At the beginning of the simulation, all processors, including the GVT master, perform barrier synchronization.

- When a message is sent, it is always marked with the current time quantum of the sender.
- During the *i*th time quantum, the *j*th simulating processor keeps track of $S_{i,j}$, the number of messages sent by itself, $OT_{i,j}$, the smallest timestamp among all messages sent in this quantum, and $\{R_{k,j}\}$, an array of integers indicating the number of messages received marked with a time quantum k.
- At the end of the time quantum i (i.e., when it is time to move to the next time quantum), the *j*th processor reports $S_{i,j}$, $OT_{i,j}$, $\{R_{k,j}\}$, as well as i, the current time quantum index, and $T_{i,j}$, the local virtual time, to the GVT master. As an example, suppose that a processor, in the time quantum *10*, has sent *6* messages, with a smallest timestamp being *35.3*, and received *9* messages. Among these *9* received messages, *3* were sent at time quantum *10*, *2* at time quantum *9*, *4* at time quantum *8*. Besides, the local virtual time at the end of the time quantum *10* is *40.8* (the local virtual time must take into account the messages received in this time quantum). The processor will then report *10*, *40.8*, *35.3*, *6*, and *{3, 2, 4}* to the GVT master.
- The GVT master maintains three arrays. The first one is $\{TM_i\}$, for the number of transient messages from each time quantum. The second one, $\{MVT_i\}$, records the smallest timestamp among all messages sent from each quantum. The last one, $\{LVT_j\}$, stores the local simulated time of each processor.
- When receiving a report message for processor j, the GVT master first obtains the time quantum index i, and then updates its three arrays according to the following rules:
  - TM_i += $S_{i,j}$
  - For each k in $\{R_{k,j}\}$, TM_k -= $R_{k,j}$
  - If ($OT_{i,j} < MVT_i$) $MVT_i = OT_{i,j}$
  - $LVT_j = T_{i,j}$
- The new GVT value is the minimum among $\{LVT_j\}$, and $\{MVT_i\}$ for such i that *TM_i is a non-zero*. More formally:
  - $GVT = min(min_i(MVT_i | TM_i > 0), min_j(LVT_j))$

In the above version of the algorithm, the report message contains an array of integers each of which denotes the number of received messages marked with the corresponding time quantum. In the DSIM implementation, a maximum length is imposed on this array, by limiting the reporting of received messages to the oldest time quantum active at this processor (the smallest k such that $R_{k,j}>0$), and possibly several others that follow immediately. By doing so the correctness of the algorithm is not affected; the only effect may be that the GVT will be conservatively computed. This happens in an infrequent case in which at most a limited number of time quanta corresponding to the maximum length of the received time quantum array allowed in the report message can be removed from consideration at the end of a time quantum. In practice, an array of size 2 to 4 gives the optimal performance. Hence, only 2 or 4 such numbers need to be sent to the GVT master, no matter how many processors are used.

The correctness of the algorithm can be proved by showing that any event message m_1 received after a GVT value *gvt* is received must have a timestamp $ts(m_1) \geq gvt$.

Proof. We prove this property by contradiction using induction.

Let's assume that $ts(m_1)<gvt$ and i_1 is the processor that sent this message at time quantum s_1. Let j_1 be the time quantum from which the last report message sent by processor i_1 that was received by the GVT master before gvt value was obtained. It cannot be that $ts(m_1)\geq T_{i1,j1}$, since $T_{i1,j1}$ is taken into account in computing gvt, so $T_{i1,j1}\geq gvt$. Neither it could be that $s_1\leq j_1$, since then $ts(m_1)$ would be reflected in the *MVT* value corresponding to s_1 , contradicting our assumption that $ts(m_1)<gvt$. Hence, $s_1>j_1$ and $ts(m_1)< T_{i1,j1}$, so there must be another message m_2, sent by a different processor i_2, which caused a rollback on processor i_1, and satisfies $gvt<ts(m_1)<ts(m_2)$, as rollback never affects the events with the same or earlier timestamps than the timestamp of the rollback message itself. From that it follows that this message also satisfies $s_2>j_2$, where s_2, j_2 are analogs of s_1, j_1.

By induction, let's assume that there is a message m_k, sent by processor i_k, such that $gvt>ts(m_k)$ and $s_k\leq j_k$, where j_k denotes the latest time quantum on processor i_k that is included in computing gvt and s_k is the time quantum at which message m_k was sent. It cannot be that $ts(m_k)\geq T_{ik,jk}$, since $T_{ik,jk}$ is taken into account in computing gvt, so $T_{ik,jk}\geq gvt$. Hence, $ts(m_k)< T_{ik,jk}$, so there must be another message m_{k+1}, sent by a different processor i_{k+1}, which caused a rollback on processor i_k, and satisfies $gvt<ts(m_k)<ts(m_{k+1})$. We define s_{k+1}, j_{k+1} as analogs of s_k, j_k. For message m_{k+1}, it cannot be that $s_{k+1}\leq j_{k+1}$, since then $ts(m_{k+1})$ would be reflected in *MVT* value corresponding to s_{k+1}, contradicting our conclusion that $ts(m_{k+1})<gvt$.

Hence, by induction we conclude that our assumption that $ts(m_1)<gvt$ implies that there is an infinite sequence of messages with time stamps smaller than *gvt,* which contradicts the basic premise that each simulation can generate only a finite number of messages in the finite simulation time *gvt*.□

3. Implementation Details

In this section, several techniques related to memory usage are described to provide the readers with an understanding of the inner workings of DSIM. Although some similar techniques may have been proposed in other systems, a combination of them used in DSIM defines the exact design under which the experiments presented in this paper have been run, thereby allowing replication of those experimental results.

3.1. Local Fossil Collection

In Time Warp, a processed event becomes a fossil when its timestamp is smaller than the GVT, and the operation of releasing memory allocated for such events is called *fossil collection*. Events cannot be immediately released after having been processed because of possibility of rollbacks. Hence, they must be kept in memory until fossil collection is performed. Usually, there are two ways of maintaining these processed events. The first approach uses a single processed event list for all LPs (Logical Processes) on one processor. The main drawback is that, when rollbacks occur on some LPs, it becomes difficult to keep the entire list sorted. If the list becomes unsorted, then the entire list has to be scanned in order to know which events are subject to fossil collection. GTW used an on-the-fly technique [23] to partially solve this problem, by checking events with local minimum timestamps and ignoring other events. However, there may exist some processed events with timestamps earlier than GVT that cannot be reclaimed by this technique.

The other approach to maintaining processed events is to keep a separate processed event list on each LP. The problem of unsorted lists no longer exists because it is always the head (the latest one) of the processed event list that needs to be rolled back first. However, it introduces another problem. As explained in [5], it is costly to search through all those processed event lists during fossil collection, especially when the number of LPs is high. ROSS [5] solved this problem by grouping LPs into kernel processes, thereby helping to reduce the number of processed event lists. Another technique [24] is to sort these processed event lists further by their tails (the earliest ones) so that those lists with a tail larger than the GVT can be completely skipped.

DSIM adopts the separate processed event list approach, in which fossil collection is not carried out when the GVT is updated. Instead, each LP checks if the GVT has been updated *before* it is about to process an event. If so, it will then compare the earliest processed event with the GVT. Only if the GVT has been recently updated and if the earliest processed event is earlier than the GVT will the LP invoke the fossil collection procedure. Otherwise fossil collection will be bypassed. Although this technique does not decrease the number of operations, it improves the locality of memory references, since the event memory released in the fossil collection procedure can be immediately reused in the processing of the new event (if there are new events to be scheduled).

Local Fossil Collection comes with its own drawbacks. First, if an LP suddenly becomes inactive after a highly active period and before the GVT is updated, it will have no new events to process and consequently will not be able to perform fossil collection. Second, delaying fossil collection until an LP is processing an event may increase memory usage, as more processed events will stay in the memory.

3.2. Event Management

Event allocation and deallocation are often operations that are the most frequently executed during discrete event simulation. Low-level system calls cannot be directly used, since such calls cannot be guaranteed to finish in a constant time. To minimize the cost of these operations, DSIM creates a customized event allocator for each event type. The rationale is that in C/C++, for objects of the same type, the memory footprint is always the same. The actual amount of memory used may vary from one object to another, but any extra memory must be explicitly obtained, and this is not a responsibility of the simulator.

Since event allocators only handle events of equal sizes, they can pre-allocate a number of memory buffers in a free buffer pool. To handle a request for a new event, the event allocator simply retrieves one buffer from the free buffer pool. If the pool is empty, it will acquire more buffers for it, using the low-level memory allocation function. When an event is to be released, its buffer is returned to the free buffer pool.

In DSIM, an unprocessed event becomes a processed one after it has been processed, and, conversely, a processed event becomes an unprocessed one after it has been undone. To save memory and to avoid unnecessary memory operations, however, both events are represented by the same data structure, with a flag denoting the status of the event. Therefore, within this single event data structure, there are two data blocks, one for the processed event and the other for the unprocessed one. DSIM is capable of overlying one over the other, to save memory further, as these two blocks are never needed at the same time during execution.

DSIM adopts the direct cancellation approach proposed by Fujimoto [25] for intra-processor events and extends it to inter-processor events. When a new inter-processor event is to be created and scheduled for an LP in another processor, the event is packed into a positive message. A stub event, containing only the timestamp of the new event, the message identifier, and the receiving LP identifier, is left behind to preserve the event dependency. The stub event, treated in the same way as other intra-processor events, will be inserted into the dependent list of the event that scheduled the new inter-processor event.

When receiving a positive message, an LP must unpack the message to restore the inter-processor event, and store the sender LP identifier and the message identifier into an input queue. To cancel an inter-processor event, an anti-message is created from the corresponding stub event. When the receiver LP sees the anti-message, it will check its input queue for presence of an inter-processor event to be canceled by comparing the message identifier and the sender LP identifier. In this way, the output queue can be completely avoided, while the input queue is still retained, but only for inter-processor events.

4. Programming Interface

DSIM comes with a simple programming interface that hides many implementation details while providing much freedom for programmers in making design decisions. To build a simulation, a DSIM programmer must first define the types of events, and then implement the LPs that comprise the simulation, and finally complete several auxiliary functions according to certain rules.

4.1. Event Declaration

The following syntax is used to create a new event type named *new_event_t*:

```
typedef tw_event_t < positive_data_t, anti_data_t, type_id>
new_event_t;
```

In the above statement, *positive_data_t*, the positive data type, is the type of the data stored in the unprocessed event, while *anti_data_t*, the anti-data type, is the type of the data stored in the processed event. The third parameter *type_id* is the identifier of the event type. It must be different for each event type so that the LP can determine what event has happened by checking this field of the event.

4.2. State Saving versus Reverse Computation

The positive data type is decided by the simulation model semantics. The determination of the anti-data type is more difficult. Normally, it is determined by what data are needed in order for the event to be reversible. DSIM supports two styles of undoing events, one is traditional state saving (infrequent state saving is not supported) and the other is based on *reverse computation* [26]. If the former is taken, the anti-data must basically store any change made by the processing of an event. For reverse computation, generally, much less information needs to be stored in the anti-data. For example, if an ordinary random number generator is used during the event processing, then the anti-data must contain the random seed before the event arrives, which can be written back to the random number generator if the event is to be reversed. This is traditional state saving. In contrast, a reversible random number generator [26] can be used, which can go back to the original state after the reverse function is called.

4.3. Implementing LPs

An LP class must be derived from the base class *tw_lp*. There are five functions of *tw_lp* that can be overridden: *Start*, *Stop*, *Process*, *Undo*, and *Commit*.

The *Start* and *Stop* functions are called when the simulation is started and stopped, respectively. The *Process* function processes an unprocessed event, while the *Undo* function revokes a processed event and brings the LP to the previous state.

The *Commit* function is called when a given processed event is to be reclaimed. This gives the LP an opportunity to perform irreversible operations, such as printing to the standard output or some others I/O actions. If the event contains some pointer to allocated memory, it is also in this function that the memory can be released.

The *Commit* function is useful for debugging as well. In a PDES program, most programming errors would cause the total number of processed events to be different when the number of processors varies. Locating the first event that is incorrect is difficult, since a processed event may be rolled back later. A customized *Commit* function would allow only those committed events (processed events earlier than the GVT) to be printed out. By comparing the outputs of a program running on different numbers of processors, incorrect events can be quickly identified.

4.4. Auxiliary Functions

Programmers must also implement several auxiliary functions in order for the simulation to work properly. These functions include *id_to_proc*, which returns the identifier of the processor where a given LP resides, and *id_to_lp*, which converts the identifier of the given LP to the pointer to this LP. In DSIM, each LP has a unique integral identifier that is used to address them. For instance, to schedule an event for an LP, the identifier of that LP must be provided. DSIM will then call the *id_to_proc* function to determine whether the destination LP is in the same processor. If so, it will call the *id_to_lp* function to obtain the address and forward the event. Otherwise, DSIM will simply pack the event into a message and then send it to the destination processor.

Another function that must be implemented by the programmer is *tw_main*. In this function, a predefined DSIM simulation engine must first be initialized. This simulation engine is then used to create a set of LPs and to initialize them one by one. After setting the parameters of the simulation engine and calling its *Setup* function, the *Run* function, which contains the main scheduler loop, must be called to run the simulation. The simulation terminates after the return from the *Run* function.

4.5. Shared/Distributed Memory

Although DSIM was designed for distributed memory clusters, it supports shared-memory machines as well. The reason for dual mode support is that shared-memory machines, while limited by the number of processors, may be a more convenient platform to program, debug, and test PDES programs. The only difference between these two modes is in the creation of LPs. In the distributed mode, the program is only responsible for creating LPs that are assigned to the current processor, while in the shared-memory mode the program must create all LPs no matter which processor they are assigned to. The GVT algorithm proposed by Xiao et al. [27] is adopted for the shared-memory version. As integral identifiers rather than addresses, are used to address LPs, the programming can completely ignore the differences between these two modes until they have to create LPs, which can be accomplished by merely a few lines of code.

4.6. Artificial Rollback

Another feature of DSIM is that it provides a special debug mode to facilitate debugging PDES programs. If the program is compiled with a macro *DSIM_AR*, artificial rollbacks will be introduced even though the simulation is executed on a single processor. This is achieved by using an error-prone priority queue that does not always find the earlier unprocessed event. Instead, with an adjustable probability, this queue may retrieve a randomly chosen, arbitrary event in the queue. Therefore, LPs will receive out-of-timestamp order events even when there is only one processor. The advantage of this technique is that errors are now reproducible, so the sequence of rollbacks remains the same every time the program is executed. In contrast, a PDES program with a regular priority queue running on multiple processors tends to produce different orders of execution, and therefore errors may appear or disappear during different runs, making the program extremely hard to debug.

5. Experimental Results

Experiments have been performed on two different clusters, in order to evaluate the performance and scalability of DSIM with respect to the PHOLD model [9]. The first set of experiments was carried out on a cluster of 40 nodes, where each node is an IBM Netfinity server with two 700-MHz Intel Pentium III processors, half of which connected by fast Ethernet and the other by Gigabit Ethernet.

In the PHOLD model [9], each event stays at an LP for an exponential time and then departs to one of four nearest neighbors randomly chosen. In all experiments present here the number of initial events per LP is always *16*. LPs are organized into a two dimensional *X* by *Y* grid, where *X* is the number of columns and *Y* is the number of rows. Let *N* denote the number of processors used.

Strip partition by columns is selected so that each processor has a sub grid of *X/N* by *Y* to simulate. One metric, the ratio of remote events (or inter-processor events), is of particular importance in asserting the performance of any Time Warp simulations. For the PHOLD model that is partitioned by columns, the ratios of remote events can be easily deduced from *X* and *N* as follows. Only two columns in each processor, the leftmost and the rightmost, may generate events that must be sent to a different processor. However, on average only 1 out of 4 such events produced by LPs on these two columns will actually depart. Therefore, the fraction of remote events is *N/(2X)*.

It must be noted that strip partition does not result in fewest remote events. It is block partition, which divides the entire grid into two-dimensional tiles, that leads to the lowest percentage of remote events when *X*=*Y*. However, one goal of the performance studies here is to demonstrate the scalability of DSIM with different ratios of remote events, so block partition is not implemented.

It is also worth to note that although the PHOLD model may seem to be a toy example, it is indeed difficult to be parallelized, because there is no lookahead, and because the event granularity is extremely low. Some real-world models, such as PCS networks [28], exhibit similar event behaviors as PHOLD, only with higher ratios of local events and coarser event granularity. Successful parallelization of the PHOLD model may shed some light on this class of real-world applications.

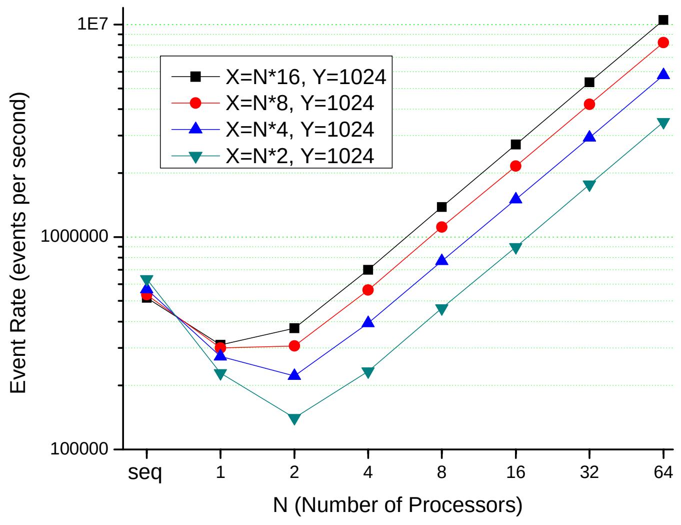

Figure 1. Performance of DSIM on a Netfinity Cluster with Fixed Percentages of Remote Events

The first set of experiments was designed to demonstrate how DSIM scales with different percentages of remote events (Figure 1). The number of processors marked on the horizontal axis does not count the GVT master. Here, the problem size was increased linearly as more processors were used. The percentages of remote events ranged from 3.125%, to 6.25%, 12.5%, and 25%. The actual percentages of remote events were slightly higher, since there were anti-messages. It is evident from Figure 1 that the performance of DSIM drops as remote events increase, due to the extra time needed for sending and receiving messages.

In Figure 1, DSIM is also compared with a sequential discrete event simulator that uses the same simulation engine but with facilities to handle rollbacks and to exchange message being removed. The overhead of parallelization becomes manifest when comparing this simulator with the sequential execution of DSIM, which is at least twice slower. More interestingly, with decreasing numbers of columns, the performance of the sequential simulator improves while that of DSIM on one processor drops. The former happens because of shrinking memory footprints with fewer LPs. The latter results from the use of a particular parameter called *event batch* which controls the number of consecutive events that can be processed in a single batch. With more remote events, this parameter must be decreased, and therefore the performance declines.

The performance gap between 1 processor and 2 processors hints the overhead of remote events. The event-processing rate may grow or drop when 2 processors are used as opposed to 1, depending on the percentage of remote events. However, in all cases starting from 2 processors, DSIM maintains almost linear increases in event rates, implying an excellent scalability of the simulator.

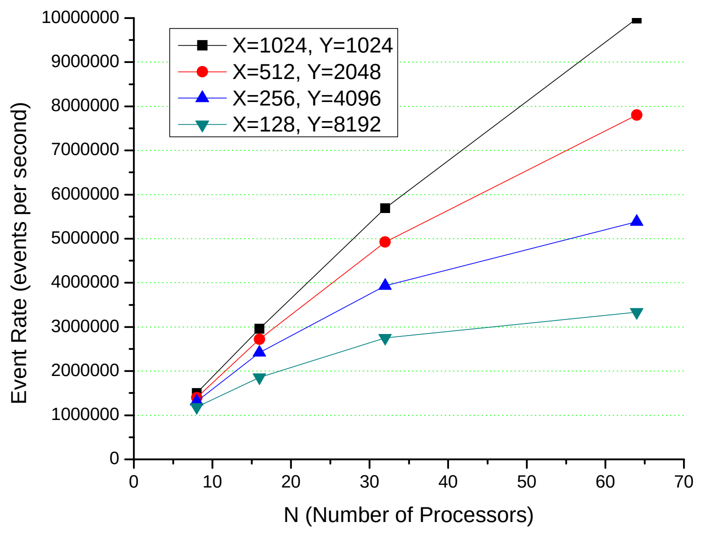

Figure 2. Performance of DSIM on a Netfinity Cluster with Fixed Problem Sizes

In the second set of experiments running on the same cluster, the number of LPs simulated was fixed but the numbers of processors varied (Figure 2). Four configurations, namely, 1024 by 1024, 512 by 2048, 256 by 4096, and 128 by 8912, give different ratios of remote events. As these results indicate, DSIM is capable of simulating about 10 millions events per second using 64 processors, with 3.25% remote events (the 1024 by 1024 configuration). When the ratio of remote events increases to 25% (the 128 by 8192 configuration), DSIM is still able to process more than 3 million events per second. For this configuration, the improvement of using 64 processors over 32 processors is minor, but with 32 processors the ratio of remote events drops from 25% to 12.5% as in this set of experiments the problem size is fixed.

What is the speedup on 64 processors? Unfortunately, for the presented performance studies, the term *speedup* loses its precise meaning. All four configurations, each of which contained 1 million LPs and 16 million events shown in Figure 2 could not be executed on less than 8 processors due to the memory requirement (even if the efficient sequential simulation engine were to be used), so it was impossible to measure how fast a sequential simulation of the same problem size would be. If, on the other hand, the data presented in Figure 1 are to be used to derive speedups, the results would be biased against parallel execution, since models of different sizes are simulated, and larger sizes mean slower sequential speedup, as evident in Figure 1. Even so, the speedup of parallel execution on 64 processors of a 1024 by 1024 grid versus efficient sequential execution of a much smaller size (16 by 1024) is still 20.3.

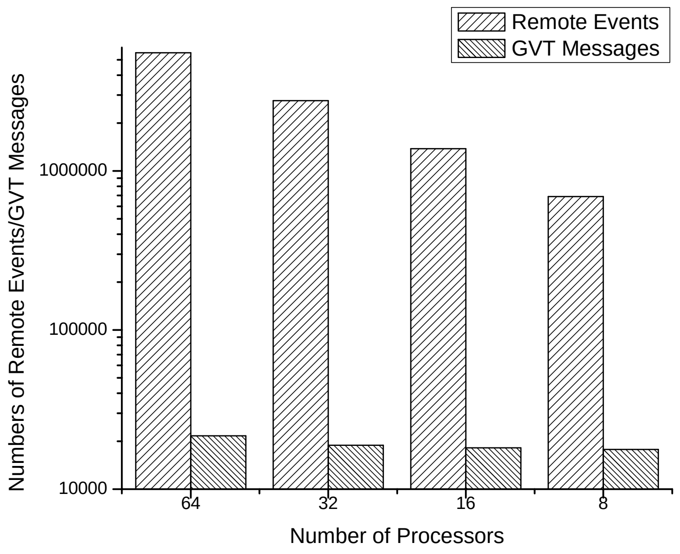

Figure 3. Numbers of Remote Events/GVT Messages in the 1024 by 1024 configuration

Figure 3 shows the numbers of remote events and GVT messages in the 1024 by 1024 configuration. The numbers of GVT messages stay roughly the same for different numbers of processors as they are proportional to the product of the execution time and the number of processors, which in turn is proportional to the workload. This is because the frequency of GVT updates is controlled by the width of time quanta. In all experiments presented in this paper, this width is 0.1-0.2 seconds, meaning that the GVT value can be updated 5-10 times per second. Notice the logarithmic scale used in the vertical axis – the ratios of GVT messages to remote events are 0.39%, 0.68%, 1.3%, and 2.6% respectively. These numbers illustrate the extremely low overhead of the TQ-GVT algorithm even with a high GVT update frequency.

Another set of experiments was run on an Alphaserver cluster consisting of 750 nodes, connected by a Quadrics interconnection network, located at Pittsburgh Supercomputing Center. This cluster was ranked 34th on the top 500 supercomputer list as of November 2004. Each node is a SMP with 4 1-GHz processors and 4 Gbytes of memory. Figure 4 depicts the event processing rates of DSIM on up to 1024 simulating processors. The numbers alongside each data point denote the number of extra processors that were used for GVT masters. It has been empirically determined that a GVT master in the TQ-GVT algorithm can drive as many as 128 processors without degrading the performance noticeably (whether more processors can be supported is yet to be decided). Therefore, 2, 4, and 8 intermediate GVT masters were introduced for 256, 512, and 1024 processors respectively.

The number of columns increases as more simulating processors are added, to maintain constant 6.25% remote events. A performance curve similar to those in Figure 1 is observed in Figure 4. In the 1024 processor case, which actually used 1033 processors, 67,108,862 LPs were simulated, yielding an event processing rate of 228 million events per second, and a speedup of 296 (again, this speedup is somehow unfair to parallel execution since it is impossible for any single CPU to execute a simulation this large). Since each LP was assigned 16 events initially, at any moment during the simulation there were totally 1,073,741,824 events.

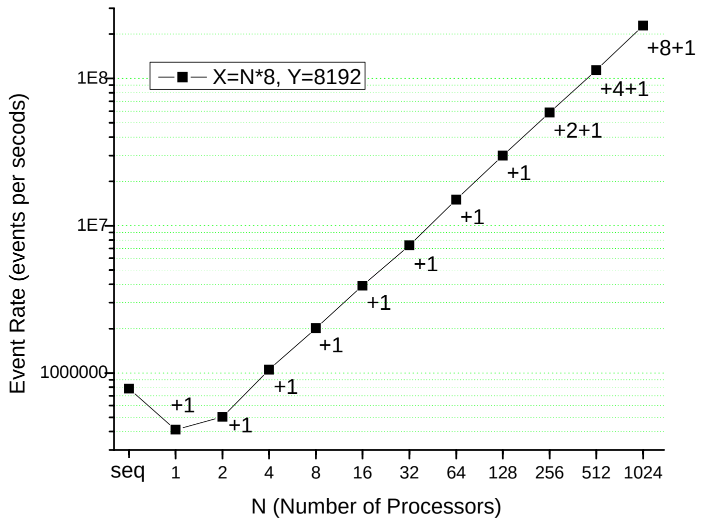

Figure 4. Performance of DSIM on the PSC Cluster with Fixed Percentages of Remote Events

6. Conclusion and Future Works

As clusters become more prevailing, more and more PDES applications will be executed on this type of parallel computers. DSIM was developed under such an assumption. The major difficulty of porting PDES applications to clusters is designing an efficient and scalable GVT algorithm. The Time Quantum GVT algorithm adopted by DSIM meets these requirements. Furthermore, various improvements, such as Local Fossil Collection and efficient event management, enable DSIM to run with an unprecedented speed of 228 million events per second. As the same time, the programmability of DSIM has been an equally important design consideration to ensure that a programmer can quickly get familiar with the simulator.

DSIM is freely available at http://www.cs.rpi.edu/~cheng3/dsim. More simulations will be implemented to verify its performance for various applications. The Time Quantum GVT algorithm will continue to be improved to enable Time Warp simulations on tens of thousands or even millions of processors. Moreover, as DSIM is an open source project, it is hoped that it will provide a standard simulation platform for researchers to implement and test various PDES algorithms.

Acknowledgement

The largest simulations were carried out on the National Science Foundation Terascale Computing System at the Pittsburgh Supercomputing Center.

Reference

1. Baezner, D., G. Lomow, and B. Unger, *Parallel simulation environment based on time warp.* International Journal in Computer Simulation, 1994. **4**(2): p. 183.
2. Jefferson, D., B. Beckman, F. Wieland, L. Blume, and M. Diloreto. *Time warp operating system*. in *Proceedings of the eleventh ACM Symposium on Operating systems principles*. p. 77-93.
3. Jefferson, D.R., *Virtual time.* ACM Transactions on Programming Languages and Systems, 1985. **7**(3): p. 404-25.
4. Das, S., R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. *GTW: a time warp system for shared memory multiprocessors*. in *Proceedings of Winter Simulation Conference, 11-14 Dec. 1994*. 1994. p. 1332-9. Lake Buena Vista, FL, USA.
5. Bauer, D., G. Yaun, C.D. Carothers, M. Yuksel, and S. Kalyanaraman. *ROSS.Net: Optimistic parallel simulation framework for large-scale Internet models*. in *Proceedings of the 2003 Winter Simulation Conference: Driving Innovation, Dec 7-10 2003*. 2003. p. 703-711. New Orleans, LA, United States.
6. Martin, D.E., P.A. Wilsey, R.J. Hoekstra, E.R. Keiter, S.A. Hutchinson, T.V. Russo, and L.J. Waters. *Redesigning the WARPED simulation kernel for analysis and application development*. in *Proceedings 36th Annual Simulation Symposium (ANSS-36 2003), 30 March-2 April 2003*. 2003. p. 216-23. Orlando, FL, USA.
7. Mascarenhas, E., F. Knop, and V. Rego. *ParaSol: a multithreaded system for parallel simulation based on mobile threads*. 1995. p. 690. Arlington, VA, USA.
8. Kim, H.K. and J. Jean. *Concurrency preserving partitioning (CPP) for parallel logic simulation*. 1996. p. 98. Philadelphia, PA, USA.
9. Fujimoto, R.M. *Performance of time warp under synthetic workloads*. in *Distributed Simulation. Proceedings of the SCS Multiconference, 17-19 Jan. 1990*. 1990. p. 23-8. San Diego, CA, USA.
10. Baldwin, R., M.J. Chung, and Y. Chung. *Overlapping window algorithm for computing GVT in Time Warp*. in *11th International Conference on Distributed Computing Systems (Cat. No.91CH2996-7), 20-24 May 1991*. 1991. p. 534-41. Arlington, TX, USA.
11. Bauer, H. and C. Sporrer. *Distributed logic simulation and an approach to asynchronous GVT-calculation*. in *6th Workshop on Parallel and Distributed Simulation (PADS92). Proceedings of the 1992 SCS Western Simulation MultiConference on Parallel and Distributed Simulation, 20-22 Jan. 1992*. 1992. p. 205-8. Newport Beach, CA, USA.
12. Bellenot, S. *Global Virtual Time Algorithms*. in *Proceedings of the SCS Multiconference on Distributed Simulation, Jan 17-19 1990*. 1990. p. 122-127. San Diego, CA, USA.
13. Choe, M. and C. Tropper. *An Efficient GVT Computation Using Snapshots*. in *CSMA98*. 1998. p. 33-43.

14. D'Souza, L.M., X. Fan, and P.A. Wilsey. *pGVT: an algorithm for accurate GVT estimation*. in *Proceedings of 8th Workshop on Parallel and Distributed Simulation, 6-8 July 1994*. 1994. p. 102-9. Edinburgh, UK.
15. Preiss, B.R. *The Yaddes Distributed Discrete Event Simulation Speficiation Lnaugage and Execution Environment*. in *Proceedings of the SCS Multiconference on Distributed Simulation*. 1989. p. 139-144.
16. Samadi, B., *Distributed Simulation, Algorithms and Performance Analysis*, in *Computer Science Department*. 1985, University of California, Los Angeles.
17. Tomlinson, A.I. and V.K. Garg. *An algorithm for minimally latent global virtual time*. in *1993 Workshop on Parallel and Distributed Simulation, 16-19 May 1993*. 1993. p. 35-42. San Diego, CA, USA.
18. Das, S.K. and F. Sarkar. *A hypercube algorithm for GVT computation and its application in optimistic parallel simulation*. in *Proceedings of Simulation Symposium, 9-13 April 1995*. 1995. p. 51-60. Phoenix, AZ, USA.
19. Mattern, F., *Efficient algorithms for distributed snapshots and global virtual time approximation.* Journal of Parallel and Distributed Computing, 1993. **18**(4): p. 423-34.
20. Steinman, J.S., C.A. Lee, L.F. Wilson, and D.M. Nicol. *Global virtual time and distributed synchronization*. in *Proceedings 9th Workshop on Parallel and Distributed Simulation (ACM/IEEE), 14-16 June 1995*. 1995. p. 139-48. Lake Placid, NY, USA.
21. Srinivasan, S. and P.F. Reynolds, Jr. *Non-interfering GVT computation via asynchronous global reductions*. in *Proceedings of 1993 Winter Simulation Conference - (WSC '93), 12-15 Dec. 1993*. 1993. p. 740-9. Los Angeles, CA, USA.
22. Perumalla, K. and R. Fujimoto. *Virtual time synchronization over unreliable network transport*. 2001. p. 129. Lake Arrowehead, CA, USA.
23. Fujimoto, R.M. and M. Hybinette, *Computing global virtual time in shared-memory multiprocessors.* ACM Transactions on Modeling and Computer Simulation, 1997. **7**(4): p. 425-46.
24. Vee, V.-Y. and W.-J. Hsu. *Pal: a new fossil collector for time warp*. in *Proceedings 16th Workshop on Parallel and Distributed Simulation, 12-15 May 2002*. 2002. p. 35-42. Washington, DC, USA.
25. Fujimoto, R.M., *Time warp on a shared memory multiprocessor.* Transactions of the Society for Computer Simulation, 1989. **6**(3): p. 211-239.
26. Carothers, C.D., K.S. Perumalla, and R.M. Fujimoto, *Efficient optimistic parallel simulations using reverse computation.* ACM Transactions on Modeling and Computer Simulation, 1999. **9**(3): p. 224.
27. Xiao, Z., F. Gomes, B. Unger, and J. Cleary. *A fast asynchronous GVT algorithm for shared memory multiprocessor architectures*. in *Proceedings 9th Workshop on Parallel and Distributed Simulation (ACM/IEEE), 14-16 June 1995*. 1995. p. 203-8. Lake Placid, NY, USA.
28. Carothers, C.D., R.M. Fujimoto, and L. Yi-Bing. *A case study in simulating PCS networks using time warp*. 1995. p. 87. Lake Placid, NY, USA.