Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

SYNCSIM: A SYNCHRONOUS SIMPLE OPTIMISTIC SIMULATION TECHNIQUE BASED ON A GLOBAL PARALLEL HEAP EVENT QUEUE

Sushil K. Prasad
Zhiyong Cao

Computer Science Department
Georgia State University
Atlanta, GA 30303, U.S.A.

ABSTRACT

We developed and implemented two highly optimized optimistic discrete event simulation techniques based on an efficient and scalable *Parallel Heap* data structure as a global event queue. The primary results are (i) the design of an optimistic simulation algorithm, namely *SyncSim*, which does not rely on traditional state and message saving data structures, but employs only one backup state per state variable, (ii) a demonstration, through implementation of *SyncSim*, of an optimistic technique which overcomes the two main mutually conflicting and unbounded overheads of the existing optimistic simulation algorithms: *SyncSim* bounds the additional space requirements to just one copy per state variable and drastically limits the number of rollbacks encountered. Furthermore, *SyncSim* beats the highly optimized traditional simulator *simglobal* on a wide variety of large networks on an Origin-2000 computer. The algorithm *SyncSim* could form a basis for a good parallelizing engine attachable relatively easily to an existing serial simulator.

1 INTRODUCTION

We designed and implemented two optimistic simulation algorithms for discrete event simulation. The first algorithm, called *simglobal*, is a highly optimized traditional optimistic simulator (Prasad and Naqib 1995; Prasad 2000a). The state vector is partitioned into logical processes (lps), and these lps asynchronously simulate and advance their local clocks while coordinating with other lps using messages. An lp receiving message before its local clock rolls back. To enable rollback mechanism, the simulation model is modified. Each lp has an attached queue to save its past states and all its input messages and the output messages produced by executing some of the input messages (Jefferson 1985; Fujimoto 1990). Global virtual time (GVT), which is upper-bounded by the earliest message in the system, keeps track of the progress of the entire system. Past states and messages before GVT are periodically discarded. The second algorithm, called *SyncSim,* requires only one backup copy of the state variable, which means that it gets rid of the usual data structures associated with state variable or logical processes including the queues for saving past states, and those for input and output messages. We adopt *Parallel Heap* (Prasad and Sawant 1995; Deo and Prasad 1992) as the global event queue in both simulation algorithms. By varying the node size, the *Parallel Heap* can exploit the available parallelism of a network system. *Parallel Heap* has been demonstrated to be highly efficient not only theoretically but also in practice on large priority queues with even fine-grained parallel accesses. It has been effectively employed in development of theoretical algorithms and practical implementations of parallel simulations (Prasad 1990; Prasad 1993; Prasad and Junankar 2000).

We thoroughly tested these two simulation schemes on both regular torus networks and on other ill-behaved networks on our Origin-2000, which is a shared-memory machine with 24 CPUs, 4 GB main memory, and a NUMA architecture with hypercube interconnect employing CRAY links.

The primary results are two-fold. (i) The design of an optimistic simulation algorithm, namely *SyncSim*, which does not rely on traditional state and message saving data structures, which we believe to be the first of its kind. (ii) A demonstration, through implementation of *SyncSim* on a commercial parallel computer, of an optimistic technique which overcomes the two main unbounded overheads of the existing optimistic simulation algorithms: *SyncSim* bounds the additional space requirements to just one copy per state variable and drastically limits the number of rollbacks encountered, better than even *simglobal*. We believe the latter result also to be uniquely demonstrated. Furthermore, *SyncSim* beats the highly optimized traditional simulator *simglobal* in actual performance on a wide variety of large networks.

The simulator *SyncSim* outperforms simulator *simglobal* both in speedup and rollback. We also compared the performance of *Parallel Heap* as global event queue with a

global *heap* and found that *Parallel Heap* is more efficient in simulating large network than *heap*. For small to medium size systems which generate moderate number of events/messages, one can also choose among various concurrent priority queues including calendar queue (Brown 1988), concurrent heaps (Rao and Kumar 1998), and concurrent skew heaps (Jones 1989). In addition, because of the special character of *Parallel Heap*, the first element of the root node is always has the earliest message/event. The GVT is just set to the time field of that element, so it is even easier to obtain GVT than pGVT and other algorithms (D'Souza et al. 1994).

The algorithm *SyncSim* could form a basis for a good parallelizing engine (Prasad 1997; Prasad 2000a) due to the simplicity of its pluggable and modular data structures and algorithms coupled with its good performance. Fujimoto and Tsai (1993) have demonstrated that simulators developed in a variety of languages, including SIMSCRIPT, GPSS, GASP and MODSIM, can be translated into C language simulators conforming to a generic model of discrete event simulator and then parallelized. Their optimistic algorithm employed the usual data structures at each logical process, including multiple backup states to enable rollback to a recent past, as well as sorted queues of input and output messages to facilitate rollbacks. However, their scheme was too elaborate, with significant search and maintenance overheads and complicated optimistic control code, to the point of being not so efficient. Nicol and Heidelberger (1996) have demonstrated a conservative-algorithm-based parallelization, but their scheme depends on system characteristics such as look-ahead information. Our strategy, SyncSim, presented here is optimistic in nature, and represents improvements over the previous work of Fujimoto and Tsai's group. Learning from their past experience, the primary foci here are simplicity and efficiency. Some other related research are High-Level Architecture based joining of multiple simulators to work together as a way of potentially exploiting some parallelism, albeit through a layer of generic HLA protocols (Ferenci 2000, Riley 1999), and object-based general parallelization of parallel programs as pursued in (Back and Turner 1995), among others.

The rest of these paper is organized as follows: first, *Parallel Heap* data structure is introduced which forms the key data structure around which the two simulation algorithms are designed (Section 2); second, both *simglobal* and *SyncSim* algorithm are described (Section 3 and 4, respectively); third, the implementation details are explained and the simulation results are analyzed (Section 5); finally, conclusions are drawn (Section 6).

2 *PARALLEL HEAP*

Parallel Heap is the first heap-based data structure to have efficiently implemented a scalable parallel priority queue on an EREW parallel random access machine (Prasad 1990; Deo and Prasad 1992). Employing p processors, a *Parallel Heap* allows deletion of O(p) highest priority items and insertion of O(p) new items, each in O(log n) time, where n is the size of the *Parallel Heap*. Practically, *Parallel Heaps* are shown to be highly efficient parallel priority queues even with a large number of fine-grained parallel accesses (Prasad and Sawant 1995), and has been successfully employed previously for simulating fine-grained systems such as VLSI logic circuits (Prasad and Junankar 2000).

Parallel Heap employs the same concept as the conventional priority heap except that in *Parallel Heap*, each node contains up to r items for $r >= 1$, and the r highest priority items are always in the root node. It allows synchronized deletions of these r or fewer items from the node and insertion of up to $2r$ new items in each insert-delete cycle. All r items of a node must have values less than or equal to the items at its children (*Parallel Heap* condition). Only the last node in the heap may have fewer than r items. All the items in a *Parallel Heap* node are kept in sorted order. The new items to be inserted are first sorted. These new items, starting from the root node, go down toward the bottom, merging with each intervening nodes and carrying down the larger item each time. Similarly, after deletion of $k <= r$ items of the root, k items are brought to the root from the bottom. The root node may not satisfy the *Parallel Heap* condition any more. A delete-update process in employed to update the *Parallel Heap* to make it satisfy the *Parallel Heap* condition.

Using *Parallel Heap* for simulation involves two steps:

- Simulation phase (also referred to as think phase), in which the deleted items are processed/simulated possibly generating new items.
- Insert-delete phase, in which the newly generated items are inserted to the *Parallel Heap* and the smallest r items are deleted for simulation in the next cycle.

The above procedures can be implemented in a pipelined fashion by executing a delete-update process followed by up to two insert update processes at each level, as follows.

PerformInsertDelete( ) cycle:

1. Process the update process at the odd levels of the *Parallel Heap*, in parallel, and move them down to the even levels.
2. Think/Simulate using the r deleted items. Sort the newly created items
3. Merge the newly created items with the root node. Delete the smallest r for the subsequent think phase. Get substitute items from the last node, if needed, and initiate update processes at the root node (level 0).
4. Process and move the update processes at the even levels down to the odd levels.

3 OPTIMISTIC SIMULATOR *SIMGLOBAL*

In Prasad and Naqib (1995), we had compared a previous version of *simglobal* on a shared-memory machine with *simlocal,* its counterpart with local event queues, and with *localDist*, which is load-distributed version of *simlocal* wherein output messages are inserted to a random local event queue. All event queues employed were parallelized calendar queues. *Simglobal* drastically reduced rollback frequency and beat others in speedup obtained over a variety of networks. This established the utility of a global event queue to be an effective technique to control the rollback frequency because (i) the various components of the logical network being simulated move forward in simulated time more or less in a synchronized fashion, and (ii) the computational load is balanced across all processors. In the current version, a *Parallel Heap* is employed to serve as the global message/event queue and contains a copy of the earliest messages corresponding to each lp in the logical network being simulated. Simulation proceeds in synchronized cycles by deleting and simulating *r* earliest messages at their lps. Thus, the *Parallel Heap* inherently yields a dynamic time-window comprising earliest *r* messages in each cycle (Prasad 1990; Prasad 1993; Sokol et al. 1988; Steinman 1993).

3.1 Algorithm Simglobal

Each lp of simulator *simglobal* has a single channel that contains all its messages, implemented as a time-ordered linked-list. Each executed message is attached with negative copies of its outgoing messages and with the state of the lp before the message is executed. A pointer is used to keep track of the earliest message to be executed next. This pointer gets updated whenever a message is simulated or a new message is inserted into the channel. When a new positive message is inserted into the channel of a destination lp, if a rollback is caused then the earliest message pointer is moved to this new message, the local clock and state are properly restored, and a copy of this message is inserted into the event queue.

The actual rollback execution takes place lazily (Gafni 1988), for further optimization. When the current message is being re-executed because of a prior rollback, the newly produced messages are compared with the old negative messages that were sent out. If different, both the new positive message and the old negative messages are inserted into the destination lps. In the process of the insertion of the old negative message, its copy of positive message is found. If no rollback is caused, the positive message is deleted, else it is marked *void* and the correct state and local clock are restored at the destination lp. The negative message just gets deleted after its positive message is found. When the corresponding positive message is deleted from the event queue for execution, it finds its voided copy in the channel at its lp, deletes it, and sends out the old negative message. Garbage collection of the committed messages in the channel is performed whenever an lp is accessed for simulation.

3.2 Implementation of Simulator S*imglobal* Using *Parallel Heap*

The original *PerformInsertDelete()* function in the *Parallel Heap* is modified to act as the simulation-driving program to continue while the desired *SimulationTime* >= *GVT*. As mentioned in Section 2, there are four steps in this function. Steps 1, 3, and 4 are unchanged. The only additional detail is in Step 2. The implementation of Step 2 is as follows:

2a) Divide the *r* deleted message elements from the previous Step 3 among the processors in a round robin fashion. This partition method is aimed to reduce rollbacks.

2b) After getting their messages, each processor asynchronously simulates the messages at the corresponding lps. GVT, equal to the time stamp of the earliest message deleted from GQ, is employed to perform garbage collection of earlier messages at these lps.

2c) After simulating each message, the newly generated messages are inserted to the destination lp's channel, and the earliest message of that channel is copied to the *NewItems* array.

2d) Then each processor sorts its messages in their own section. The sorted elements in each section are merged in parallel into a sorted array.

After the above step, all new messages are in *NewItems*. Then continue with the Step 3 in the *Parallel Heap PerformInsertDelete()* function. Therefore, importing the simulation code to be combined with *Parallel Heap* code is straightforward. It doesn't require any change to the original simulation code.

4 OPTIMISTIC SIMULATOR *SYNCSIM*

Both s*imglobal* and *SyncSim* employ a global *Parallel Heap* to control rollback frequency and repeatedly simulate the earliest batch of *r* messages synchronously. The batch size *r* is tuned to the level of concurrency exhibited in the simulated system (currently through sample runs, but can be adopted online). *SyncSim* additionally enforces a constant state-space overhead. It opts for spatial parallelism over temporal parallelism to avoid deeper rollbacks, which would result from the lack of multiple past states to rollback to.

4.1 Algorithm *SyncSim*

SyncSim features four key differences when compared to s*imglobal:*

(i) *SyncSim* has only one backup state per lp (yielding strict upper bound on state-space usage in an optimistic algorithm).

(ii) It does not have a sorted list of input/output messages and state queues at each lp. Instead, all messages reside in the global message queue. This enables *SyncSim* to be a good candidate for a parallelizing engine attachable relatively easily to an existing serial simulator without complex data structures at lps.

(iii) It primarily exploits spatial, not temporal, concurrency, i.e., unless the earliest message at an lp is garbage collected, no attempt is made to optimistically simulate a later message at that lp.

(iv) Since one backup state is already being used in *SyncSim,* it also uses an additional space for one message at each lp, simplifying the algorithm without asymptotically increasing state-space overhead. In order to make sure that each rollback decision is correct, *SyncSim* keeps the earliest executed message at its lp (without its copy in GQ) until it gets discarded, or gets bumped by an earlier message, at which moment it goes into the GQ. It re-executes when extracted from the GQ next time if the earliest message has been voided or discarded, and causes rollback before re-execution if lp's local clock is greater than its own time stamp. If this space is unoccupied, it stores a copy of the earliest unexecuted message among all messages that have arrived at an lp, with its original in GQ.

Note that the 1st feature alone would force each rollback in *SyncSim* to be a deep rollback to *GVT,* but in itself does not present any correctness problem. The 2nd feature, on the other hand, would mean that *SyncSim* would face rollbacks not only because of messages not arriving at an lp in non-decreasing order of time stamps, but also because each such rollback may face several subsequent "jitter" rollbacks as intermediate messages may not get re-simulated in sorted linear order of time stamps in a concurrent environment.

More severely, without the 3rd and the 4th features, there is a resulting need for rollback counters and optimistically delayed garbage collection techniques to counter two problems as briefly explained below (Prasad 2000a). With only one backup state, it is possible that an lp had multiple messages simulated creating a scenario of having a backup state s' at local clock lc' and its current state s at local clock such that $lc' < GVT < lc$. Thus, one or more already executed messages before the current GVT exist which have not been garbage collected because of the lack of a backup state at GVT. The 3rd feature, that of throttling an lp to its earliest message, eliminates this problem because the backup state is always at GVT when the earliest executed message is garbage collected. Another problem that could have occurred is that without the context available from the sorted queues of input messages as in traditional optimistic simulations, rollback decisions could be wrong or unnecessary. It is possible, for example, that an already simulated message event *e* with time stamp *t(e)*, which has not been garbage collected yet and hence is still in the global event queue, will not realize that, after *e*'s simulation, its lp rolled back to a time before *t(e)* because an earlier message arrived, and then moved ahead of t(e) because a second message with timestamp greater than *t(e)* subsequently arrived, and that a rollback to time *t(e)* and *e*'s re-execution is now essential for correctness of the simulation. As explained in 4th feature, such a situation does not arise in *SyncSim* because an already simulated message, when re-extracted from GQ, rolls back the lp if it has progressed beyond its time stamp.

Thus, the 3rd and 4th features become crucial to eliminate the problems associated with garbage collection and correctness of rollbacks introduced by the first two. Even if an lp moves forward too far, each subsequent rollback linearly and quickly reverts it toward its earliest message. This enforces a rollback length of just one each time, and simplifies garbage collection of the earliest executed message once GVT catches up to it. As experimental results show, the amount of loss of temporal parallelism is compensated amply by the exploitation of spatial parallelism in large systems.

The *voiding* method is lazily delayed after the re-execution of a message. The children of a message are voided only if the output of the re-execution is different from the previous execution. The local time of each lp is assigned to the time of the last message it simulated. The new global virtual time, GVT, is calculated as in *simglobal.* We did not experiment with other GVT algorithms to focus exclusively on optimistic control mechanisms.

Figure 1, 2, and 3 show the pseudo-codes for algorithm *SyncSim*. The *simulate()* function of Figure 1 is called on each message deleted from the *Parallel Heap* replacing Step 2b and 2c of *simglobal* algorithm. The *simulate()* function, in turn, may call *execute()* function of Figure 2 to simulate or re-simulate a message, and the *insert()* function of Figure 3 to send output messages to the destination lps. These are all constant-time steps except for the *Parallel Heap* access times, which is logarithmic (Prasad and Sawant 1995).

Symbols employed are as follows:

- *lp:* logical process
- *GQ*: Global Queue
- *t(m):* time stamp of message
- *lc:* local clock
- *GVT:* Global Virtual Time

```
Simulate (Message m at lp i)

if (m voided) /* if simulated already, rollback
                 already effected by insert() */ {
    - void children of m
        /* no effect if m not executed */
    - discard m, and its copy at lp i if m was
      the earliest;
}
/* m not voided */
else if (lp i's earliest message m' is simulated
          before){
    if (m' voided){
        - void descendant of m'; discard m';
          roll back lp i;
        - execute(m,i);}
    else if (t(m') <= GVT){
        - discard m'; execute(m,i);}
    else if (t(m) < t(m'))
            /* same as t(m) < lc of lp i */{
        - rollback lp i; execute(m,i);
        - m' goes to GQ;}
    else  m goes back to GQ;
}
else if (lp i's earliest message m' is not
          executed yet){
    if (m' voided){
        discard m'; execute(m,i);}
    else if (t(m) > t(m')) m goes back to GQ;
    else {
        - discard m'; /*copy of m' already in GQ*/
        - execute(m,i);}
}
else /*lp i empty */  execute(m,i);
```

Figure 1: Algorithm *SyncSim:* How to Simulate Message *m* at lp *i* Deleted from Global Queue

```
execute (message m at lp i):

1.  Backup state of lp i and simulate m;
    m becomes lp's earliest;
2.  if (t(m) = GVT) discard m;
3.  if (m being simulated first time)
        insert() children of m at their lps;
    else /* m was simulated before */
        if new output messages of m are different
        from its old outputs, void old ones and
        insert() new output messages;
       /* lazy cancellation */
```

Figure 2: Algorithm Sync*Sim:* How to Simulate or Re-Simulate Message *m* at lp *I*

Note: In Prasad (2000a), we had experimented with a preliminary version of *SyncSim,* namely Simulator *Simple*, based on a parallelized calendar queue as the global event queue, and compared it with calendar queue based version of s*imglobal*. *Simple*, despite its space constraints, was closely behind s*imglobal*. It became apparent that parallel calendar queue, which has a serial bottleneck to delete the batch of earliest message and hence is non-scalable, was limiting *Simple's* potential of exploiting spatial concurrency available in large systems. This has led to the development of *SyncSim* and s*imglobal*, based on highly scalable and efficient *Parallel Heap* data structure. Prasad (2000a) describes the detailed process of how simulator

```
insert (message m at lp i) /* Direct cancellation
      in place; so m can not be an antimessage */

1. if (lp i empty) m becomes its earliest;
else if (lp's earliest m' voided) {
    - if (m' executed before){
        - void children of m';
        - rollback lp i; }
    - m replaces m':
        {discard m'; m becomes lp's earliest;}
}
else if (t(m') <= GVT) and (m' simulated before)
    - m replaces m';
else if (t(m) < t(m')) and (m' not executed)
    - m replaces m';
else if (t(m) < t(m')) and (m' executed before){
    - rollback lp i; m becomes lp i's earliest;
      m' goes into GQ;
}

2. Original copy of m goes into GQ.
    /* the original has direct pointers from its
    parent */
```

Figure 3: Algorithm *SyncSim:* How to Handle an Output Message m Sent to lp i

Simple has been systematically developed, aided by experimentation on different versions of simulator *simglobal*; Prasad (2000b) shows the theoretical development of a series of algorithms on which simulator *Simple* and *SyncSim* are based on.

4.2 Implementation of *SyncSim* Algorithm Using *Parallel Heap*

One key implementation difference between *SyncSim* and *simglobal* is that a message, after being simulated, need to keep track its children till it is garbage collected in order to be able to void the child message directly in case of rollback. This is implemented by adding a *pointer field* to every message to store the memory address of its child. Therefore, the implementation of *Parallel Heap* needs to be modified to reflect this change. In the previous implementation of *Parallel Heap*, the actual message is physically stored in the *Parallel Heap* node, each node being an array of the *message* structure type. When updating the *Parallel Heap*, the actual message is moved from one node to another node or from one place in a node to another place in the same node, which means that the memory address of the message is changed when the *Parallel Heap* updates itself. It is very difficult for a parent message to keep track of the child messages under this *Parallel Heap* implementation. So the implementation of *Parallel Heap* is modified: instead of each node being an array of the message type, now it is changed to the array of pointer of message type. Thus, no matter how the *Parallel Heap* updates, only the pointer to an actual message moves and the actual message itself stays where it was. In this case, a parent message can always find its child messages. However, because the *Parallel Heap* maintains itself according to the

time field of the message, it becomes an overhead when the node is only an array of pointer variable to the real message. So the *Parallel Heap* node structure is further modified: each element of the array not only contains the point variable to the real message but also stores the time field of the real message. In this case, it doesn't need the indirect memory access to get the time field in updating the *Parallel Heap*.

When each participant processor gets a message from the *DeletedItems*, it does the simulation as shown in the *SyncSim* algorithm. All the newly generated items, including the original message, which cannot be simulated because an earlier message is still sitting in the node, have not being garbage collected. These messages, and those which are bumped from the node because an earlier message arrived, are inserted into the *NewItems*. All the rest are same as in the implementation of *simglobal* simulation.

5 EXPERIMENTAL DETAILS AND RESULTS

Initially, we employed these two simulators on torus networks. Next, we simulated the randomly generated networks. On each set of networks, *SyncSim* outperformed *simglobal.*

5.1 Simulation of Large Torus Networks

Each lp in the torus network has two output channels connecting to its top and right neighbor lps and two incoming channels from its left and bottom neighbor lps. The service time of each lp is randomly chosen from 0 to 5 with 10% of the lps' service times are set to 0 to test the performance of the simulators in ill-behaved situations (forcing lots of rollbacks). The total number of lps in the torus network was one million. The initial number of message was one for each lp. We choose medium event grain using an empty for-loop with 8000 iterations. Coarser grain improves the performances further.

We also varied different parameters of the *Parallel Heap* as follows:

1) The total number of processors used is divided into two parts: the number of processor used for simulation and the number of processors used for maintaining the *Parallel Heap*. If total number of processors used is *t*, number of maintenance processors is *t'*, the number of processor to do the simulation is *s,* the relationship between these three number is: if $t' < t$, $s = t - t'$; if $t' = t$, $s = t$. The best general settings are shown in Figure 4 for the networks tested: (t, s) = (1, 1), (2, 2), (4, 4), (8, 6) are the best setting between total processor used and number of participating simulation processors.

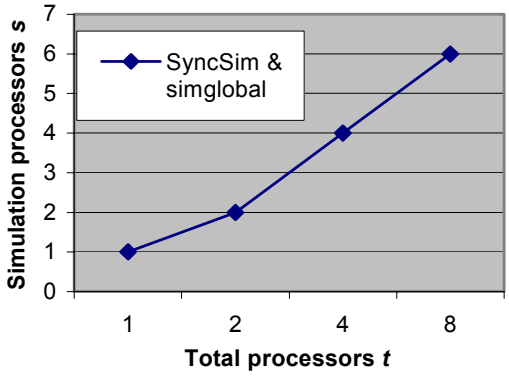

Figure 4: Total Processors used and the Number of Participating Simulation Processors

2) The node size *r* is very critical to the performance of simulator. If the node size is too small, the simulator can't exploit all the parallelism of the system. If the node size is too large, the wrong messages may be simulated too far before it is rolled back. In the experiments, we choose r = 500, 1000, 2000, 4000, and found that *r=1000* is the optimal number for simulator *SyncSim* and *r=500* is the optimal number for simulator *simglobal* (Figure 5). These optimal settings are used in the subsequent plots.

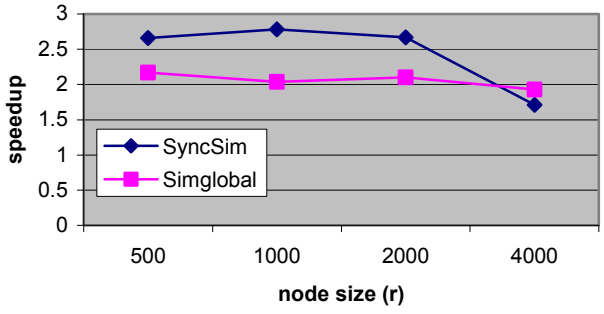

Figure 5: Relationship between Speedup and *Parallel Heap* Node Size

Simulator *simglobal* based on a local heaps for each processor with locks for insertion and deletion is implemented to compare the result. We call this simulator ***simlocal***. Figure 6 shows the speedup comparison among these three simulators. As we can see, simulator *SyncSim* out perform the rest two with speed up 4.34 using 8 processors; the simulator *simglobal* performs second with speed up of 3.54 using 8 processors.

Figure 7 shows the rollback number comparison among these three simulators.

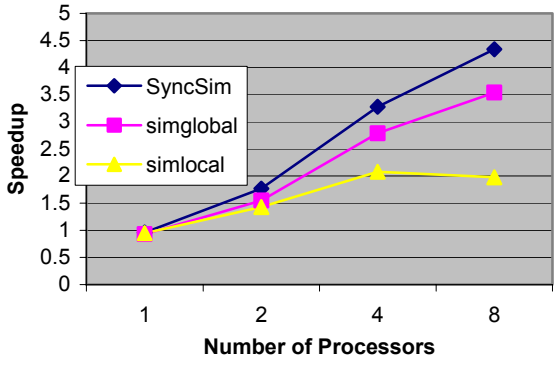

Figure 6: Speedup Comparison on Torus (Think Time: 8000)

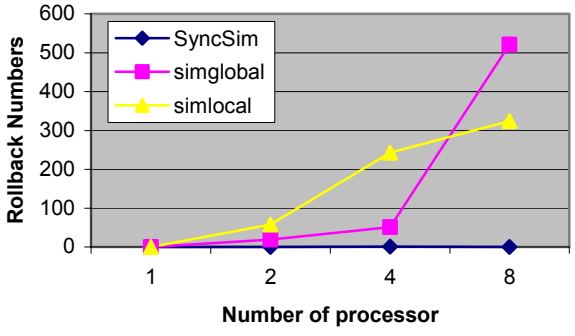

Figure 7: Rollback Comparison on Torus Network (Per Million Messages)

To our surprise, at the beginning, we found that when the number of processors used reached 8, the rollback number of *simglobal* version was higher than that of *simlocal*. Usually the rollback number should be less with global queue than that with local queue. In order to verify our result, we implemented another version of *simglobal* based on a single global heap with locks. As we expected, simglobal successfully reduced the rollback number (Figure 8). The reason the rollback number of *simglobal* exceeds that of *simloca*l when the processor number is 8 can be explained as follows: because in *simglobal*, processors don't synchronize with each other before they finish simulating the numbers of messages assigned to them in each cycle, with more processors participating in the simulation, it is possible that the simulation goes wrong much further than in the *simlocal* which synchronize each processor after each message per processor get simulated.

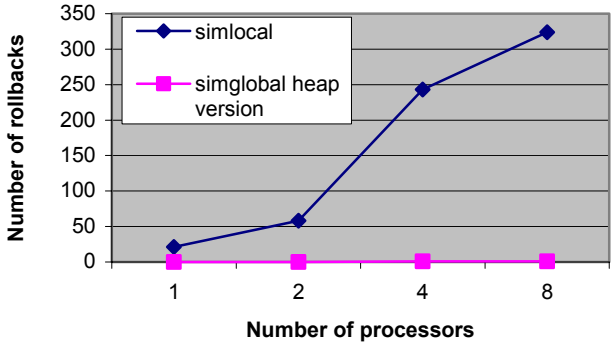

Figure 8: Rollback Number Comparison between *simlocal* and *simglobal* Heap Version

5.2 Simulation of Random Networks

Encouraged by the performance of simulator *SyncSim* and *simglobal* on regular network, we further tested the performance of these two simulators on static randomly generated network. In static randomly generated network, the two output channels and the two incoming channels of each lp are randomly generated. The total number of lps is 100K. Other initial settings were same as in torus network simulation. Figure 9 and 10 show the speedup comparison and rollback comparison between simulator *SyncSim* and *simglobal* on static random networks, respectively.

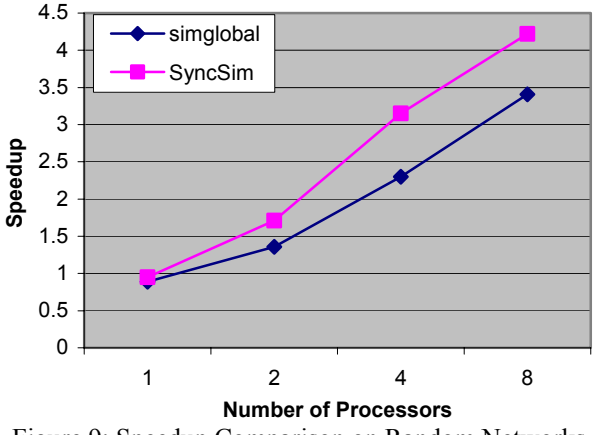

Figure 9: Speedup Comparison on Random Networks

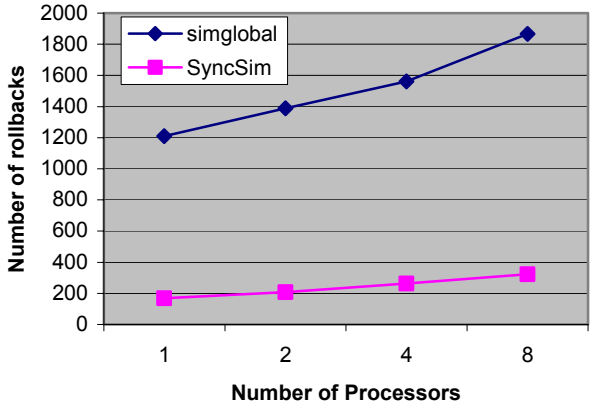

Figure 10: Rollback Comparison on Random Networks (Messages Simulated: 55000)

We find that the performance of simulator *SyncSim* and *simglobal* are very consistent on static random networks with those on torus networks, except that the rollback

numbers of both simulators *SyncSim* and *simglobal* is much higher than that on torus network.

In order to verify the utility of the *Parallel Heap* structure on large networks, we also implemented the simulator *SyncSim* and simulator *simglobal* using a single heap with locks for insertion and deletion as a global event queue on the same torus network used before. The speed up result is shown in Figure 11.

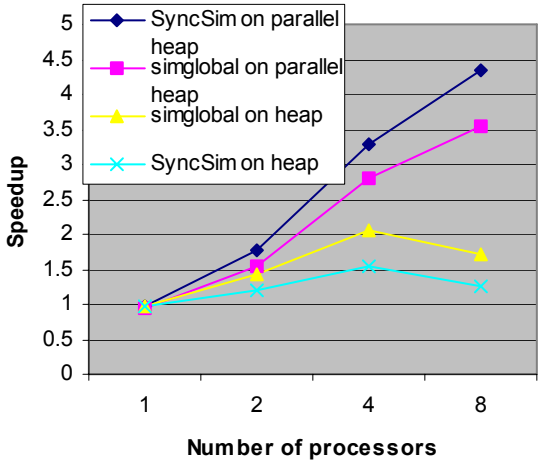

Figure 11: Comparison of Speedup between Simulator *SyncSim* and *simglobal* on Different Event Queue Data Structures.

The performance of both simulator *SyncSim* and *simglobal* decreased when the processor number reached 8 because there are so many locking and unlocking activities associated with the single heap. Therefore, the heap itself has become the bottleneck of the performance. We also find that the performance of simulator *simglobal* is better than that of *SyncSim* using single heap as a global queue. This can be explained as follows: *SyncSim* is more global queue bound as it lacks the supporting data structures at its lps. It tolerates the additional accesses by exploiting the spatial parallelism of large systems. Slow access to global queue adversely affects these advantages. On the other hand, *simglobal* improves with more frequent synchronizations afforded in the serial access heaps (after each s messages in heaps as opposed to after $r >> s$ messages in *Parallel Heaps*, where s is the number of processors simulating.).

6 CONCLUSIONS

We presented the design and implementation of an optimistic simulation algorithm, namely *SyncSim*, which does not rely on traditional state and message saving data structures, which we believe to be the first of its kind. Also, we demonstrated that this algorithm overcomes the two main unbounded overheads of the existing optimistic simulation algorithms. Algorithm *SyncSim* bounds the additional space requirements to just one copy per state variable and drastically limits the number of rollbacks encountered, better than even *simglobal*. We believe the latter result also to be demonstrated for the first time. Furthermore, *SyncSim* beats the highly optimized traditional simulator *simglobal* in actual performance on a wide variety of networks.

The performance of an optimistic parallel simulation algorithm depends on the data structure it uses. With the efficient parallel data structure, *parallel heap,* simulator *SyncSim,* is a very good candidate for parallelizing engine, attachable to an existing simulation code of a large network simulation.

REFERENCES

Back, A and S. J. Turner. 1995. Using Optimistic Execution Techniques as a Parallelisation Tool for General Purpose Computing. In *International Conference on High Performance Computing and Networking (HPCN Europe '95),* Milan, Springer, 21-26.

Brown. R. 1988. Calendar Queue: A fast O(1) priority queue implementation for the simulation event set problem. *Comm. ACM*, 31: 1200-1227.

Deo, N. and S. K. Prasad. 1992. Parallel Heap: An optimal parallel priority queue. *J. Super-computing,* 6:87-98.

D'Souza, L.M., X. Fan and P.A.Wilsey. 1994. pGVT: An algorithm for accurate GVT estimation. In *Procs. 8th Workshop on Par. and Dist. Sim*. (PADS). 101-108.

Gafni, A. 1988. A rollback mechanism for optimistic distributed simulation systems. *Procs, SCS Multiconf. Dist. Sim*., Vol.19, No.3, 61-67.

Ferenci, S. L., K.S. Perumalla, and R.M. Fujimoto. 2000. An Approach for Federating Parallel Simulators. In Pocs. PADS'00, 63-70.

Fujimoto, R. M. 1990. Parallel discrete event simulation. *Comm. ACM.* 33: 31-53.

Jefferson, D.R. 1985. Virtual Time. *ACM Trans. Prog. Lang. Systems*, 7: 405-425.

Jones, D. W. 1989. Concurrent operations on priority queues. *Comm. ACM*, 32, No. 1, 132-137.

Nicol, D. and P. Heidelberger. 1996. Parallel execution for serial simulators. *ACM Trans. Modeling and Sim.,* vol 6, No. 3, 210-242.

Prasad, S. K. 1990. Efficient parallel algorithms and data structures for discrete event simulation. Ph.D. Dissertation, Computer Science Dept., Univ. of Central Florida, Orlando.

Prasad, S. K. 1993. Efficient and scalable PRAM algorithms for discrete event simulation of bounded degree networks. *J. Parallel and Distributed Computing,* 18: 524-530.

Prasad, S. K. and B. Naqib. 1995. Effectiveness of Global Event Queues in Rollback Reduction and Dynamic Load Balancing in Optimistic Discrete Event Simulation. In *Procs. 9th Workshop on Parallel and Distributed Simulation,* Lake Placid, NY, 187-190.

Prasad, S. K. and S. Sawant. 1995. Parallel Heap: A practical parallel priority queue for fine-to- medium-grained applications on small multiprocessors. In *Proc. 17th IEEE Symp. Parallel and Distributed Processing,* San ANTONIO, Texas, 328-335.

Prasad, S. K. 1997. A Framework for Automatic Parallelization of Existing Discrete Event Simulators. In *Procs. IEEE Conference on Networking and Distributed Computing*, Kharagpur, India, Dec 5-7.

Prasad. S. K. 2000a. Practical Global-Event-Queue-based Optimistic Simulation Algorithms with One Backup State Vector and Low Rollback Overheads. In *Procs. The First International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'2000),* May 22-24, Hong Kong, Chapter 5.

Prasad. S. K. 2000b. Space-Efficient Algorithms based on Global Event Queues for Parallelization of Existing Discrete Event Simulators. In *Procs. The First International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'2000),* May 22-24, Hong Kong, Chapter 20.

Prasad, S. K. and N. Junankar. 2000. Parallelizing a Sequential Logic Simulator using an Optimistic Framework based on a Global Parallel Heap Event Queue: An Experience and Performance Report. In *Procs. 14th Workshop on Parallel and Distributed Simulation*, Bologna, Italy, 111-118.

Rao, V.N., and Kumar, V. 1998. Concurrent access of priority queues, *IEEE Trans. Comput.,* Vol. 37, No. 12, 1657-1665.

Riley, G. F., Fujimoto, R. M., and Ammar, M. H. 1999. A Generic Framework for Parallelization of Network Simulations, In *Procs MASCOTS'99,* 128-135.

Sokol, L., D. Briscoe and A. Weiland. 1988. MTW: a strategy for scheduling discrete simulation events for concurrent execution. In *Procs. Distributed Sim. Conf.*, 34-42.

Steinman, J. S. 1993. Breathing time warp. In *Procs. 7th Workshop on Par. and Dist. Sim. (PADS)*, San Diego, CA, 109-118.

Tsai, J.J. and R. M. Fujimoto. 1993. Automatic Parallelization of Discrete Event Simulation Programs, In *Procs. Winter Simulation Conference*, 697-705.

AUTHOR BIOGRAPHIES

SUSHIL K. PRASAD is an Associate Professor of Computer Science and Director of Yamacraw-GSU Embedded Software Program at Georgia State University. He has a B.Tech. from Indian Institute of Technology, Kharagpur, an M.S. from Washington State University and a Ph.D. from University of Central Florida, all in Computer Science. Prasad has participated in collaborative external grants and contracts with funding of about $3M. Prasad has carried out theoretical as well as experimental research in parallel and distributed computing and simulation, with 48 publications, and has made 3 utility patent applications and over a dozen provisional patent applications. His homepage is at <www.cs.gsu.edu/~cscskp>.

ZHIYONG CAO completed his M.S. in Computer Science from Georgia State University in 2002. Currently, he is employed in software industry in Atlanta.