Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

AN EFFICIENT AND SCALABLE PARALLEL ALGORITHM FOR DISCRETE-EVENT SIMULATION

Sushil Prasad
Math. and Computer Science Dept.
Georgia State University
Atlanta, GA 30303

Narsingh Deo
Computer Science Dept.
University of Central Florida
Orlando, FL 32816

ABSTRACT

We describe a new parallel algorithm for discrete-event simulation on exclusive-read exclusive-write parallel random-access machine (EREW PRAM). This algorithm is based on a recently developed parallel data structure, namely **parallel heap**, which when used as a parallel event-queue, allows deletion of $O(p)$ earliest messages and insertion of $O(p)$ new messages each in $O(\log n)$ time, using p processors, where n is the number of messages scheduled. Our algorithm can simulate up to p independent messages in $O(\log n)$ time - thus achieving $O(p)$ speedup. The number of processors, p, can be optimally varied in the range $1 \leq p \leq n$. To our knowledge, such a **theoretical efficiency** in a parallel simulation algorithm has been achieved for the first time.

1 INTRODUCTION

About a decade ago, the increasingly high computational requirement of sequential simulation led researchers to recognize that one recourse in coping with this problem is parallel simulation. Parallel simulation held the hope of a p-fold speedup over sequential simulation by using a p-processor parallel computer. The last decade saw active research in exploring parallel processing techniques for simulation. Several parallel algorithms have been proposed for simulation as a result. Unfortunately, none of these algorithms have fulfilled the promise of parallel simulation. That is, we do not know of any existing parallel simulation algorithm which could guarantee a p-fold speedup on a p-processor computer. The purpose of this paper is to demonstrate a theoretically efficient and optimally scalable parallel simulation algorithm for general physical systems. We present a new algorithm which, in the *worst-case*, guarantees $O(p)$ speedup while using p processors. Furthermore, if sufficient parallelism is available in a physical system which can be partitioned into n components, then up to n processors can be efficiently used. We begin with some preliminary concepts, and thereafter, introduce our work. Since our algorithm uses a completely new approach, we present the relevant background material in some detail to motivate readers.

Background

In this paper, we will be concerned with discrete-event simulations. Furthermore, we restrict ourselves to the parallelization of event-driven approach, which centers around the data structure, *event-queue*.

Any two events which access a common system variable must be simulated in the order of their times of occurrences to ensure the overall correctness of the simulation. Two such events are said to have a *data dependency relationship*. Likewise, if an event is caused by another event, they are said to have *causal dependency* relationship. Data and causal dependencies, together, impose a partial order on the set of all the events that can possibly occur in a system. An event which is not dependent on any other event through either data dependency or causal dependency is called an *independent* event. Usually, at any given moment during the course of simulating a system, there are multiple independent events available. The main thrust of parallel event-driven simulation is to identify and simulate independent events concurrently. Initial parallelization efforts of the sequential event-driven simulation, however, were stymied by the serial bottlenecks that the common data structure event-queue and the global variable simulation-clock presented. In each cycle of simulation, the event-queue could provide just one event and after that event's execution, the simulation-clock had to be advanced. Therefore, Misra(1986) commented: "We contend that the sequentiality inherent in the event-list structure is a major impediment to the widespread use of simulation."

Naturally, most of the parallel event-driven algorithms proposed so far have discarded the two shared objects of the sequential simulation – the simulation-clock and the event-queue. There are two major existing parallel event-driven schemes: *conservative* and *optimistic*. The conservative scheme, whose foundation was laid by R. E. Bryant(1977) and independently by M. Chandy(1979), suitably partitions a physical system to be simulated into several components called *logical processes* (*lp's*). Each lp simulates itself autonomously while coordinating with other lp's in the system by sending and receiving time-stamped messages. The local simulation-clocks of different lp's need not be synchronized. In the conservative scheme, each lp simulates itself and advances its local clock cautiously, and, if necessary waits, to ensure that it does not receive a message with a time-stamp less than its local clock. Such waiting, however, could lead to deadlock and would require deadlock detection and resolution. Otherwise, some deadlock avoidance schemes have to be adopted. The optimistic scheme, founded by Jefferson(1985), uses the same partitioning of the physical system as the conservative scheme does, but lets the individual lp's simulate themselves without waiting to coordinate with others. This can lead to an lp receiving a message with a time-stamp less than its local clock and necessitate a rollback to an earlier state. A rollback at an lp could cause its neighboring lp's to rollback also, thus leading to a proliferation of rollbacks. The overheads of deadlock management in the conservative schemes and of rollbacks in the optimistic schemes waste computational resources. Furthermore, these overheads are *unbounded* as they depend on the physical system being simulated. Even without these overheads, such as the recently proposed conservative scheme by Lubachevsky(1989) which avoids deadlock by repeatedly updating the global simulation clock, these existing schemes can not promise an efficient use of a multiprocessor computer. (An efficient parallel algorithm is one which yields a $O(p)$ -fold speedup on a p-processor computer over a uniprocessor computer.) This is because *none of them can guarantee an optimal task-to-processor allocation to achieve a balanced computational load.* There are a host of other parallel simulation algorithms falling somewhere between optimistic and conservative schemes with similar drawbacks. For a survey of the major parallel simulation approaches, refer to Fujimoto(1990) and chapter 2 of Prasad(1990). It seems that in the quest for novel algorithms, the efficiency issue has been lost – although there are algorithms with good average performances, there is none with a guaranteed worst-case performance. Besides, not much attention has been paid to the developments in parallel processing research in other areas which could provide new insights into designing parallel simulation methods that are theoretically sound and provably efficient.

Our Approach

We decided to reopen the basic issues which were abandoned a decade ago (Prasad 1990) and to directly attack the long-standing bottlenecks of parallel event-driven simulation: the shared event-queue and the global simulation-clock. The problem of updating the global simulation-clock is exaggerated. It can be efficiently updated in parallel in a logarithmic number of steps. Parallelizing the data structure event-queue was indeed a difficult problem. An event-queue is an example of an abstract data structure called *priority queue*. A priority queue allows deletion of the highest-priority item and insertion of new items. In the case of a sequential simulation, deletion of the highest-priority item corresponds to the deletion of the earliest event from the event-queue. A parallel priority queue should allow simultaneous deletion of several earliest events and insertion of several new events. The research on parallel priority queues, however, had not been successful prior to our recent work (Deo and Prasad 1990). Previously, the best result had been obtained by Rao and Kumar(1988) in parallelizing a conventional data structure called *heap*. Rao and Kumar could efficiently use up to $\log n$ processors of a shared-memory parallel computer on a heap, thus performing $O(\log n)$ insertions and deletions in $O(\log n)$ time. However, $\log n$-fold parallelism was very limited.

We have designed a new parallel data structure, namely a *parallel heap*, which, using p processors, 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. A parallel heap can efficiently utilize up to n processors on an exclusive-read exclusive-write parallel random-access machine (EREW PRAM) as compared to only $\log n$ processors that all the pre-existing methods can use. The data structure for parallel heap is similar to that of a conventional heap except that each of its node contains p items instead of just one item, where p is the number of processors used. Therefore, the root node of a parallel heap always contains the top p highest-priority items. For further details, readers are referred to Deo and Prasad(1990) and Prasad(1990).

With the parallel heap, the event-queue bottleneck of parallel simulation has been eliminated. In this paper, we employ the data structure parallel heap to design an efficient EREW PRAM algorithm for

discrete-event simulation. This algorithm uses the partitioning of a system into logical processes and their coordination by message passing as in the conservative parallel simulation algorithms. The parallel heap will be used to store messages prioritized by their times of occurrences (from now on, we will talk about messages instead of events - both have been shown to be equivalent concepts (Misra 1986)). This allows extraction of p earliest messages to be considered for simulation in every cycle, using p processors. If these p messages are not all independent, those which are independent are filtered out using specific techniques, and then simulated.

Section 2 describes an efficient algorithm for a general physical system. Our algorithm can simulate up to p independent messages in $O(\log n)$ time, where n is the number of logical processes in the system. The number of processors, p, can be optimally varied in the range $1 \leq p \leq n$. Section 3 analyzes the time complexity. Space complexity is $O(n^2)$. Section 4 contains some remarks on modifications and practical implementation. Section 5 contains some concluding remarks and suggests future work.

Machine Model

We implement our parallel algorithm on an EREW PRAM (Karp and Ramachandran 1989). This machine model consists of p identical processors and a shared-memory. Each processor has its own program. There is a global synchronization among the processors via a central clock. A processor can execute a simple instruction or read from or write into a shared-memory cell in one clock period. Furthermore, during any clock period, a shared-memory cell is accessed by no more than one processor for reading or writing.

2 A PARALLEL-HEAP-BASED CONSERVATIVE SIMULATION ALGORITHM

In this section, we present Algorithm ParHeapSim which is efficient for general systems. Let there be n lps, lp 1 through lp n. There exists a directed channel $c(i, j)$ from lp i to lp j if there is a possibility of lp i sending a message to lp j. Therefore, each lp can have up to $n-1$ incoming channels. Each lp i is associated with a local clock $lclock(i)$ initialized to 0, and a counter $nonempty(i)$ which is equal to the number of nonempty incoming channels at lp i. Algorithm ParHeapSim also uses a global simulation clock, GC, initialized to 0.

Assumptions: We assume that simulation of a message at an lp i requires constant amount of time, and its simulation produces only a constant number of output messages from lp i. We further assume that if messages have same times of occurrences, they can be simulated in an arbitrary order. Each lp i has a known *minimum service time* $s(i) \geq 0$ such that the simulation of a message m with time of occurrence $t(m)$ at lp i would not produce an output message before time $t(m) + s(i)$.

Algorithm ParHeapSim finds independent messages by using the following two properties:

1. The minimum time message at an lp whose all input channels are non-empty is independent. – **Property I**
2. A message m at an lp i can not affect another message m' at an lp j ($j \neq i$) before time $t(m) + s(i)$. Thus, if $t(m') \leq t(m) + s(i)$, then message m' is independent of message m. – **Property II**

Property II ensures that the earliest time message in the entire system can be simulated in each simulation cycle. This property also allows the execution of some additional messages whose time of occurrences are close to that of the earliest time message.

Algorithm ParHeapSim

Algorithm ParHeapSim uses a parallel heap containing copies of the minimum time messages of each of the channels in the logical system prioritized by their times of occurrences. Each such copy keeps a pointer to its original message. After a message m is simulated, m as well as its copy are annihilated. After initializing the parallel heap according to its description, and initializing all the clocks and the counters as described previously, Algorithm ParHeapSim then consists of repeated executions of the following steps (assuming p processors, P_1 through P_p):

1. (a) for $1 \leq i \leq p$, P_i deletes the ith earliest message from the parallel heap.
   (b) If there is more than one message pertaining a common lp among those deleted, the minimum time message of each lp is carried to step 2 - other messages are clearly not independent.
2. A value q equal to the minimum of $t(m) + s(i)$ for a message m of lp i, for all the messages from step 1, is calculated (if $t(m) \leq q$ for a message m, then m would be independent because of Property II).
3. A message m of lp i is carried to step 4 if
   (a) $nonempty(i)$ equals the number of incoming channels of lp i (Property I).

(b) $t(m) \leq q$ (among others, this surely includes the earliest time message of the entire system - Property II).

4. The messages from step 3 are simulated. The local clock of an lp i whose message m was simulated is incremented to $lclock(i) = t(m)$.

5. The global simulation clock GC is incremented to the time of the earliest message executed in step 4.

6. Those messages which are deleted in step 1 but not simulated in step 4 are inserted back into the parallel heap.

7. For each message m of input channel j at lp i simulated in step 4, if the channel j remains non-empty, the next minimum time message of channel j is inserted into the parallel heap. Else, channel j has become empty; therefore, $nonempty(i)$ is decremented.

8. For each message m of lp i executed in step 4, the output messages are sent to the destination lps. If an output message m' is sent from lp i to a destination lp j along an empty channel $c(i,j)$ (which causes $c(i,j)$ to become non-empty), then

   (a) m' is inserted into the parallel heap.

   (b) $nonempty(i)$ is incremented.

3 TIME COMPLEXITY

Note that the maximum size of the parallel heap is $n(n-1)$, one for each possible channel in a general system. Therefore, deletion or insertion of $O(p)$ messages will require $O(\log n)$ time ($\log(n(n-1)) = O(\log n)$) using p processors, $1 \leq p \leq n$. We will show that each of the steps of Algorithm ParHeapSim can be performed in $O(\log n)$ time on an EREW PRAM.

Steps 1(a), 2, 3, 4, 5, 6, 7, and 8(a) are implementable in $O(\log n)$ time in a straight forward way. Let us consider step 1(b). As an example, suppose processors P_4, P_7, and P_8 were three processors which deleted messages belonging to a common lp t. Then the processor with the minimum time message from lp t would be P_4 because, in step 1(a), P_i deletes the ith earliest message from the parallel heap. Thus, for step 1(b), determining the minimum time message of an lp t is equivalent to finding the minimum indexed processor which deleted a message of lp t in step 1(a). This can be accomplished in $O(\log p)$ time by a synchronized navigation of a binary tree from its leaves toward its root. This method utilizes the central clock synchronization among all processors in a PRAM as described next.

We attach each lp t with a full binary tree with p leaves, where p is the number of processors used (Figure 1). If p is not a power of two, this tree is chosen to have height of $\lceil \log_2 p \rceil$ so as to have at least p leaves. Each edge (i,j) from a node i to its parent j of this tree can hold a pair of integers initialized to [0,0]. We assign the ith leaf to P_i while counting leaves from left, for $1 \leq i \leq p$. If processor P_i has a message from lp t during the kth cycle of simulation for $k \geq 1$, P_i starts at the ith leaf of the binary tree at lp t. In each subsequent $\lceil \log_2 p \rceil + 1$ machine cycles, P_i executes the following steps:

> Let P_i be at node j with parent node *parent*.
> If j is not the root node then
>
> 1. P_i sets edge $(j, parent)$ to value $[i,k]$ indicating that P_i has a message of lp t in the kth simulation cycle.
> 2. P_i reads the value $[i',k']$ of the edge between the parent of node j and the sibling of node j. If $k' = k$, then another processor $P_{i'}$ is also accessing the binary tree at lp t in the current kth simulation cycle. Therefore, if ($k' = k$ **and** $i < i'$) **or** ($k' \neq k$), P_i moves on to the parent node. Else, P_i exits.
>
> else P_i has the minimum time message.

Thus, the winner processor arriving at the root of the binary tree of lp t has the earliest message of lp t.

Step 1(b) is also implementable in $O(\log p)$ time without specifically relying on the global synchronization of the central clock of PRAM. A sketch of an alternative method follows. The steps involved are

1. forming ordered pairs (i,j) at each processor P_j which deletes a message of an lp i in step 1(a).

2. sorting these pairs in $O(\log p)$ time using Cole's parallel mergesort algorithm (Cole 1988) such that $(i,j) < (i',j')$ if ($i = i'$ **and** $j < j'$) **or** ($i < i'$).

3. reassigning the kth sorted pair from the previous step to P_k.

4. marking a pair (i,j) if its assigned processor P_k finds that pair (i',j') assigned to P_{k-1} has $i \neq i'$. This is done in parallel by P_2 through P_n. Additionally, P_1's pair is also marked.

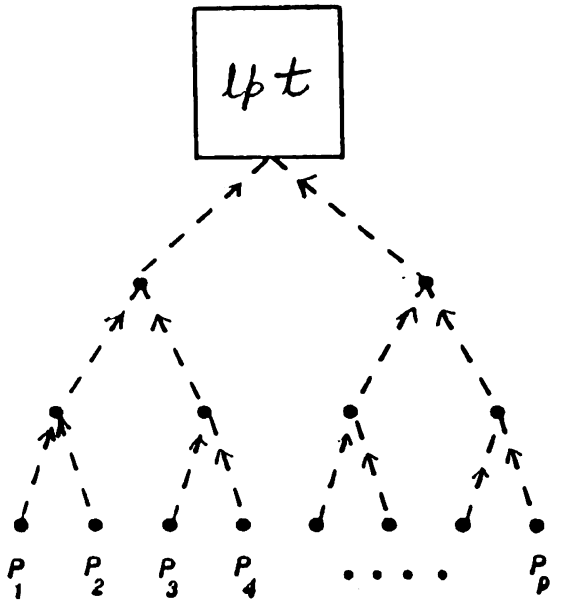

Figure 1: Binary tree at an lp for implementation of step 1(b).

It can be verified that if a pair (i, j) is marked in the last step, then P_j is the processor which deleted the minimum time message of lp i.

Thus, step 1(b) is implementable in $O(\log n)$ time (because $p \leq n$). By a similar technique, the counter $noncmpty(j)$ of a destination lp j can be properly incremented in step 8(b) of Algorithm ParHeapSim in $O(\log n)$ time.

Thus, each step of Algorithm ParHeapSim requires $O(\log n)$ time. Furthermore, the number of processors can be varied up to n, where n is the number of the logical processes. This scalability comes from the scalability of the parallel heap data structure.

4 REMARKS

Certain remarks are in order at this point. In Algorithm ParHeapSim , all we really want is to store the minimum time message of each lp into the parallel heap. This would eliminate step 1(b) and would provide an upper bound of n on the size of the parallel heap. The minimum time message of an lp, however, changes dynamically as the new messages arrive. This, in turn, would require deleting the old minimum time message from the parallel heap and inserting the current minimum time message. Parallel heap data structure, however, supports the deletion of the earliest p messages at a time, not that of any arbitrary p messages. Thus, in addition to old minimum time messages, the current minimum time message would also need to be inserted. Thus, the parallel heap could contain $n(n-1)$ messages in the worst-case, one for each channel of the logical system. To avoid these details in Algorithm ParHeapSim , we simply stored the front message of each channel into the parallel heap without affecting the $O(\log n)$ time bound on the simulation cycle.

Since the purpose of Algorithm ParHeapSim is to demonstrate a theoretically efficient parallel simulation algorithm for general systems, Algorithm ParHeapSim is not an optimized algorithm. Several improvements are possible for speeding Algorithm ParHeapSim in a practical implementation. In addition to using step 3 for filtering the independent messages, one can exhaustively test the independence of the earliest $\sqrt{p \log n}$ messages deleted in step 1 in $O(\log n)$ time (Prasad 1990). For a bounded in-degree logical system, the technique employed in Lubachevsky's algorithm (Lubachevsky 1989) is portable to Algorithm ParHeapSim as well (Prasad 1990). New techniques for finding independent messages, as and when found, can also be incorporated in step 3 of Algorithm ParHeapSim .

Even with these modifications, a genuine criticism of Algorithm ParHeapSim would be that since it considers only p earliest messages in each simulation cycle all of which might not be independent, some processors will remain unutilized. Two remedies are possible.

1. In step 1(a) of Algorithm ParHeapSim , kp earliest messages can be deleted for some suitable constant k without affecting the cycle time of $O(\log n)$. This would allow more messages to be tested for independence and, thus, increase processor utilization.

2. To fully exploit the parallelism available in a physical system, those message which satisfy Property I mentioned in the beginning of Section 2 can be identified and stored in a separate parallel heap READY. Step 1(a) would then delete p message from the original parallel heap and an additional p messages from the parallel heap READY. The messages deleted from the parallel heap READY need not be tested for independence. This approach is described in detail in (Prasad 1990). The simulation cycle time remains bounded by $O(\log n)$.

Finally, we have ignored the initialization times in our algorithms. This is because of our underlying assumption that the cost of initialization is insignificant compared to that of actual simulation of complex and large systems for extended period of simulated time. Of course, one can investigate the parallelization of the initialization tasks also.

5 CONCLUSION

Compared to all the pre-existing parallel algorithms for simulation, the algorithm presented in this paper has a distinct advantage of balanced load among all the processors. Such a load balancing has been possible due to the underlying data structure, parallel heap. The parallel heap is also responsible for ensuring that an entire system is simulated in such a way that no subcomponent of the system lags too far behind the others. This key property has been achieved by extracting from the parallel heap the p earliest messages in the entire system when using p processors. These p earliest messages are more likely than other messages to be independent of one another. This property would also be useful for obtaining frequent snapshots during the course of simulation to collect statistical data.

While most of the conservative parallel algorithms for simulation are plagued by deadlock avoidance, or detection and resolution overheads, our algorithm is deadlock-free. This is ensured by the fact that the earliest message in the entire system being simulated is always independent, and, therefore, it is executed in every simulation cycle. The global simulation clock, therefore, is continually incremented.

In contrast to the pre-existing parallel algorithms for simulation, which can not claim any definitive worst-case time bound, we can precisely state that Algorithm ParHeapSim requires $O(\log n)$ time for each simulation cycle, where n is the number of logical processes into which a system can be partitioned. In each simulation cycle, our algorithm simulates up to p messages, where p is the number of processors used. The availability of sufficient parallelism in the system being simulated (i.e., the number of independent messages consistently available in each simulation cycle) would determine how many processors should be used. When sufficient parallelism is available, Algorithm ParHeapSim is efficient for $1 \leq p \leq n$. The yardstick for efficiency is the standard sequential event-driven simulation algorithm which can execute an event/message every $O(\log n)$ time. Whereas, Algorithm ParHeapSim can simulate $O(p)$ messages every $O(\log n)$ time on an EREW PRAM.

There is plenty of scope for future work. A few are listed below.

1. Implementing our algorithm on a commercially available machine such as the BBN Butterfly GP-1000. Such an endeavor, while being a major implementation challenge in itself, would also have to face the problems of porting a PRAM algorithm to a real machine, such as memory contention and processor synchronization.
2. Improving the proposed algorithm and/or adopting it to specific applications such as battlefield management simulation and VLSI logic simulation.
3. Improving the underlying data structure parallel heap to allow efficient deletion of arbitrary items, not just the p top-priority items. Such an upgrade would simplify Algorithm ParHeapSim and would also open up other application areas where a parallel heap could be employed more comfortably than now, such as some shortest-path algorithms where the shortest paths change dynamically.

As a final note, apart from the proposed conservative algorithm for parallel simulation, a more practical parallel-heap-based optimistic algorithm has also been developed which actively reduces the rollback frequency and guarantees $O(\log n)$ -time simulation cycle (Prasad 1991). This algorithm is also expected to reduce the memory requirement for past states and messages.

REFERENCES

Bryant, R. E. 1977. Simulation of packet communication architecture computer systems. Tech. Rep. MIT-LCS-TR-188, Massachusetts Inst. Tech., Cambridge, MA.

Chandy, K. M., V. Holmes, and J. Misra. 1979. Distributed simulation of networks. *Computer Networks* 3 (Feb.): 105-13.

Cole, R. 1988. Parallel mergesort. *SIAM J. Comput.* Vol. 17, no. 4, (Aug.):770-85.

Deo, N. and S. Prasad. 1990. Parallel Heap. In *Procs. Intl. Conf. Parallel Process.* Vol. III, (Aug.): 169-72.

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

Jefferson, D. R. 1985. Virtual time. *ACM Trans. Prog. Lang. Systems* 7 (July): 405-25.

Karp, R. M. and V. Ramachandran. 1989. A survey of parallel algorithms for shared-memory machines. To appear in *Handbook of Theoretical Computer Science*, Amsterdam: North-Holland.

Lubachevsky, B. D. 1989. Efficient distributed event-driven simulations of multiple-loop networks. *Comm. ACM* 32 (Jan.): 111-31.

Misra, J. 1986. Distributed discrete-event simulation. *Computing Surveys* 18 (March): 39-65.

Prasad, S. 1990. *Efficient parallel algorithms and data structures for discrete-event simulation.* Ph.D. Diss., Computer Science Dept., Univ. Central Florida, Orlando, (Dec.).

Prasad, S. 1991. A scalable and efficient optimistic algorithm for parallel discrete-event simulation. To appear in *Procs. SIMTEC.* Orlando, FL.

Rao, V. N. and V. Kumar. 1988. Concurrent access of priority queues. *IEEE Trans. Comput.* 37 (Dec.): 1657-65.

AUTHOR BIOGRAPHIES

SUSHIL PRASAD has done his Ph.D. in Computer Science from University of Central Florida, Orlando, in 1990. Currently, he is an Assistant Professor at Georgia State University, Atlanta. His research interests are parallel algorithms and data structures, parallel simulation, graph algorithms, and complexity theory.

NARSINGH DEO is the Millican-Chair Professor of Computer Science at the University of Central Florida, Orlando. He has authored four textbooks and over 80 research papers. His research interests are parallel processing, combinatorial algorithms, and graph theory.