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

THE COST OF TERMINATING SYNCHRONOUS PARALLEL DISCRETE-EVENT SIMULATIONS

Vasant Sanjeevan
Marc Abrams

Departmant of Computer Science
Virginia Tech
Blacksburg, Virginia 24061-0106

ABSTRACT

Simulation models use many different rules to decide *when* to terminate. However parallel simulation protocols generally use a single, simple rule: each logical process terminates when it reaches a predefined time. This paper proposes seven algorithms by which an arbitrary termination condition can be mechanically added to a non-terminating parallel simulation using a conservative-synchronous protocol. Informal arguments about the performance of each algorithm are made, and the arguments are confirmed through measurement of the Bounded Lag protocol using torus networks on a shared memory multiprocessor. Conservative-synchronous protocols are shown to be inherently efficient in the experiments considered; adding a termination algorithm requires limited additional memory and increases the wall clock time required for simulation by at most an amount proportional to the number of logical processes. Furthermore, a simulation model with any stable termination condition requires negligible overhead for the termination algorithm, a property that appears to be unique to synchronous protocols.

1 INTRODUCTION

Most parallel discrete-event simulators can handle only a single termination condition: each logical process constituting the simulation terminates when its local simulation time equals or exceeds a value T. To apply parallel simulation protocols to any simulation model, algorithms must be developed to use any arbitrary termination rule. This poses a quandary for parallel simulation, because some termination conditions may be so costly to evaluate that they may cause a parallel simulation protocol to require more wall clock time to execute than a sequential simulation.

A *termination condition*, denoted C, is a boolean valued function whose domain is a subset of all simulation attributes. A simulation can terminate at any simulation time t such that the values of all attributes required to evaluate C are known at time t and the termination condition evaluated using the attribute values is *true*. Termination conditions are either *stable* or *non-stable*. A stable condition remains *true* once it becomes *true* (Chandy and Misra 1988). An example of a stable condition is, "the number of jobs processed by the simulation at simulation time t exceeds 100,000." An example of a non-stable condition is, "the number of jobs processed by the simulation at simulation time t equals 100,000."

Detecting when a non-stable condition holds requires evaluating the condition at every simulation time at which a logical process changes the value of any simulation attribute in the domain of the termination condition. At worst, the number of evaluations can grow exponentially in the number of attributes in a simulation model. Therefore it could be the case that a terminating parallel simulation requires more wall clock time to execute than a sequential simulation requires.

Our previous work proposes algorithms to terminate optimistic (i.e., time warp) and conservative-asynchronous (i.e., Bryant-Chandy-Misra) simulation protocols, and concludes that optimistic protocols have an inherent advantage over conservative-asynchronous protocols when an arbitrary termination condition is used (Abrams and Richardson 1991, Richardson 1991). In particular, both stable and non-stable termination conditions can be implemented in a straightforward manner with a negligible increase in memory requirements with optimistic protocols. However conservative-asynchronous protocols can only handle stable conditions with a negligible increase in memory requirements. To handle non-stable conditions, conservative-asynchronous protocols either require unbounded amounts of storage or they require the addition of the same state-saving mechanism present in optimistic protocols.

This paper presents seven termination algorithms for conservative-synchronous protocols. The most general of these algorithms applies to non-stable conditions and increases the additional storage required for termination to an amount proportional to the number of attributes used in a simulation model. This implies that the storage requirements of a terminating conservative-synchronous protocol can be competitive with the storage requirements of a terminating optimistic protocol.

Nicol has shown that under certain assumptions, the execution time required by a conservative-synchronous protocol is within a constant factor of optimal (Nicol 1990). In addition, conservative-synchronous protocols are simpler to implement than optimistic protocols. In addition to these arguments in favor of conservative-synchronous protocols, this paper presents arguments that the cost in execution time of adding termination to the Bounded Lag protocol (Lubachevsky 1989) simulating a torus queueing network model is constant for stable termination conditions and is proportional to the number of events executed between synchronization barriers for non-stable conditions. These conclusions are confirmed through measurements.

The next section defines the *simulation termination problem* and shows that this problem is more complex than the classic parallel program termination problem. Section 3 presents algorithms that work for *any* sequential or parallel simulation protocol and sequential or parallel computer architecture. Section 4 then refines the algorithms of Section 3 into seven algorithms specifically suited for conservative-synchronous protocols. Section 5 analyzes the algorithms in two ways, by analysis and by experimentation, to estimate the cost of the termination algorithms of Section 4 on the torus queueing network benchmark. Section 6 presents conclusions.

2 THE SIMULATION TERMINATION PROBLEM

In the *simulation termination problem*, we are given a non-terminating simulation and a termination condition. Condition C is guaranteed to be *true* when evaluated using the values of simulation model attributes at *some* value of simulation time. The problem is to state an algorithm that modifies the non-terminating simulation to:

P1. find a value of simulation time, denoted by t, such that function C evaluated using simulation attribute values at time t has value true,

P2. report the value of each simulation output measure at time t, and

P3. stop execution of the simulation and return control to the operating system.

This problem formulation is based on our belief that in a commercial parallel simulation system, one would like the user to specify a simulation model without worrying about termination (hence the simulation is non-terminating), and then separately specify a variety of termination conditions. The simulation system should automatically superimpose the termination condition on the non-terminating simulation.

Detecting when condition C is true is difficult for four reasons. First, the termination condition in general is a function of simulation attributes that are private to two or more logical processes that have some degree of asynchrony. Second, the termination condition may constrain the times at which they hold, for example, specifying the "first time" that an event occurs. Third, evaluating the termination condition may be more costly than performing the simulation itself. Finally, the condition is generally not stable; therefore at worst the termination condition must be reevaluated each time any LP increases its clock.

Related Work

There are many papers that address the classical parallel program termination problem; Mattern provides an overview of the literature (Mattern 1987). The classic termination problem differs from the simulation termination problem in four ways. First, in the classic problem, "termination detection" means detecting a stable condition, whereas many interesting simulation termination conditions are non-stable. Second, a termination condition must be a conjunct of predicates, each using variables private to one, unique logical process. However, a simulation termination condition generally is a function of simulation attributes, each private to one, unique logical process; the two examples given in Section 1 illustrate this point. Third, the program is idle when the termination condition is *true*, whereas the simulation termination problem presumes that the simulation is non-terminating. Finally, after the termination time t is determined, simulation programs require the values of a set of output measures to be evaluated using the attribute values at time t; in general t lies in the past of each logical process. No analog to recovering old attribute values to evaluate output measures exists in the classic termination problem.

3 GENERAL TERMINATION ALGORITHMS

This section presents termination algorithms that make no assumptions about the simulation protocol and computer architecture used to execute a simulation program; therefore they apply to sequential or parallel simulation using optimistic or conservative and synchronous or asynchronous protocols on sequential or parallel architectures.

The key to developing algorithms without relying on these assumption is to view the simulation termination problem as a type of search problem. Let a denote a set whose elements each correspond to a simulation time at which a simulation attribute required to evaluate termination condition C changes value. Let n denote the number of simulation attributes required to evaluate C. Each element of a is an n-tuple containing the values of all simulation attributes required to evaluate C at the corresponding simulation time. A simulation program computes set a. Furthermore, a contains an infinite number of elements, because the simulation is non-terminating.

3.1 Solution of Problem P1

The definition of a is related to the definition of a space-time rectangle, in which space corresponds to the set of simulation model attributes (Chandy and Sherman 1989). Figure 1 illustrates a portion of a space-time rectangle. In the figure, attribute a_1 changes value at time t_0, both a_0 and a_2 change value at time t_1, and so on. Part P1 of the simulation termination problem from Section 2 is equivalent to searching the space-time rectangle for *any* time $t \in \{t_0, t_1, t_2, ...\}$ at which condition C, evaluated using the attribute values at time t, yields *true*.

We propose two termination algorithms (Abrams and Richardson 1991):

Exhaustive Termination Algorithm: Evaluate $C(t)$ at each simulation time represented by set a in ascending time order until $C(t)$=*true*.

Non-exhaustive Termination Algorithm: Choose any subset of a such that the subset contains a time for which $C(t)$=*true*. Evaluate $C(t)$ at each simulation time represented by the chosen subset in ascending time order until $C(t)$=*true*.

One can construct cases to show that exhaustive termination sometimes requires less wall clock time than non-exhaustive termination to identify a simulation time at which $C(t)$ is *true*, however the average case performance is that non-exhaustive termination requires less wall clock time. The challenge in using non-exhaustive termination is choosing a subset of a that contains a time t for which $C(t)$=*true* before execution of a simulation begins. However such a subset can always be chosen before execution for a stable condition. Consider Figures 2 and 3, which characterize a stable and a non-stable termination condition, respectively. From Figure 3, non-exhaustive termination works for *any* subset of a that contains no element corresponding to a simulation time that is an upper bound for the simulation times corresponding to all elements in set a. Examples of subsets of a that work for stable termination conditions are to evaluate the termination condition every tenth time that an attribute changes value, and to evaluate the termination condition every five seconds of wall clock time.

Rules that evaluate the termination condition at fixed intervals have a disadvantage in that the optimal evaluation frequency is generally dependent on the input data to the simulation. Too frequent an interval devotes many processor cycles to evaluating C and can degrade the performance of a parallel simulator. Too infrequent an interval (i.e., once every 10 hours) will allow a simulation to run for a long time after its termination condition becomes true.

3.2 Solution of Problem P2

Once a simulation time t for which the termination condition is true has been found, simulation output measures using the values of attributes at time t must be reported. We propose that the non-terminating simulation store in memory the values of all attribute values required for output measures for all simulation times greater than or equal to the largest value of simulated time at which the termination condition has been evaluated. The advantage of this proposal is that when termination time t has been identified, the attribute values required for output measure computation at time t are available in memory. The disadvantage of the proposal is that the storage required grows monotonically with the difference, denoted D, between the largest simulation clock value of all logical processes and the value of simulation time at which the termination condition was last evaluated. This is why conservative-asynchronous algorithms require at worst unbounded memory for termination. Optimistic protocols need not save any attribute values that can be recomputed, thereby saving memory space at a cost of increased execution time; this is explained by Richardson (Richardson 1991). Conservative-synchronous protocols have an inherent advantage over asynchronous protocols because they bound the simulation time difference D and hence bound the amount of memory required.

3.3 Solution of Problem P3

An earlier paper proposed three methods to stop execution of a simulation: requesting the operating system to kill all logical processes (dissociative termination), forcing the simulation to go backwards in time (retrospective termination), and allowing the simulation to go forwards in time (prospective termination) (Abrams and Richardson 1991). Dissociative termination requires unbounded memory, and prospective termination only works for stable termination conditions. Hence retrospective termination is the method of choice in a general purpose parallel simulation system. Retrospective termination requires the roll back and re-execution mechanism of time-warp (Chandy and Sherman 1989); however Section 4 shows that the needed mechanism can be added to conservative-synchronous protocols in a straightforward manner.

4 TERMINATION ALGORITHMS FOR SYNCHRONOUS PROTOCOLS

The remainder of the paper will consider Lubachevsky's Bounded Lag simulation protocol, shown in Figure 4 (Lubachevsky 1989). We augment the protocol with a termination condition C. The algorithm proceeds synchronously in iterations. It can, however, simulate synchronous as well as asynchronous systems. Each logical process (LP) of the simulation executes this algorithm in parallel and synchronously with other LP's.

Let each of the N LP's comprising a simulation be denoted as $LP_1, LP_2, \ldots, LP_N$. T[i] is the earliest time of an event in the input queue of LP_i. B denotes a quantity called the *bounded lag* of the algorithm and is an upper bound on the difference between the simulation times of events being processed by all LP's in the current iteration. SV is a stop variable which is set to *true* when the termination condition becomes true. The algorithm assumes that the delay for an event to propagate from any node to its neighbor is positive.

We can categorize termination algorithms based on three criteria. First, is the evaluation of C done between barriers 1 and 2 or between barriers 2 and 3? Second, is the evaluation of C itself sequential or parallel? Third,is the evaluation of C exhaustive or non-exhaustive? Based on these criteria, we propose the following mnemonics to classify termination algorithms:

E -	evaluation of C is exhaustive
N -	evaluation of C is non-exhaustive
S -	evaluation of C itself is sequential
P -	evaluation of C itself is parallel
Q -	evaluation of C is done between barrier 1 and barrier 2
T -	evaluation of C is done between barrier 2 and barrier 3
R -	algorithm rolls back to the previous barrier before evaluating C

Not all combinations are interesting. We cannot have exhaustive evaluation between barriers 1 and 2 because we cannot have events being inserted into the shared queue while C is being computed using it. Also the parallel non-exhaustive evaluation between barriers 2 and 3 is not interesting as event processing usually takes more time than computing the Floor; therefore computing C in parallel with event processing is less likely to be a bottleneck.

We propose seven algorithms to solve P1 of finding a termination time t: NST, NSQ, NPQ, EST, ESTR, EPT and EPTR. The first three algorithms are instances of the non-exhaustive termination algorithm proposed in Section 3.

4.1 Algorithm NST

<u>Add following statement S10:</u>

```
if (i=2) SV:= evaluate C(t); /*only LP_2 evaluates C */
```

Using this algorithm to evaluate C imposes a negligible overhead for simple global termination conditions because the evaluation is done by only one of the LP's between barrier 2 and barrier 3 in parallel with the assignment to Floor, which is done by another LP. The evaluation of C itself, however, is done sequentially.

4.2 Algorithm NSQ

<u>Add the following process to the simulation which participates in the synchronization barriers of the *LP*'s</u>

```
while (not SV){
  synchronize;          /* barrier 1 */
  SV:= evaluate C(t);   /* use attributes stored after  */
                        /* previous iteration           */
  synchronize;          /* barrier 2 */
  synchronize;          /* barrier 3 */
}
```

In this algorithm, there is an additional operating system process dedicated to evaluating termination condition C. This process runs in parallel with the other LP's comprising the simulation. After each event processing iteration, attributes in the domain of C as well as those required to compute output measures are copied into memory after the LP's pass barrier 2 and the LP's proceed on to the next iteration of the loop. This imposes an extra memory overhead for the algorithm. The LP which evaluates C uses these attribute values to evaluate C while the other LP's are doing their event processing. If C evaluates to *true*, it sets the stop variable SV to *true*. In comparison with algorithm NST, this algorithm will perform an extra iteration of the simulation before termination is detected. This algorithm also imposes a negligible overhead to evaluate C. However, it may be more advantageous to use this algorithm if the evaluation of C becomes a bottleneck for Algorithm NST, because the Bounded Lag protocol usually takes more time to perform event processing than it takes to compute the Floor.

4.3 Algorithm NPQ

This algorithm differs from Algorithm NSQ only in that the evaluation of the termination condition C is itself done in parallel. For example, a combining tree could be used if the operators used in evaluating C form an abelian group (Lakshman and Wei); in this case algorithm NSQ would require O(N) time, while algorithm 3 requires O(log N) time to evaluate C. Parallel computation of C is justified if the time required to evaluate the termination condition is much larger than the time that LP_1 spends doing its event processing. However, there are overheads involved in parallelizing the computation that may negate the benefit of parallel evaluation of C.

The following four algorithms are instances of the exhaustive termination algorithms proposed in Section 3.

4.4 Algorithm EST

This alogrithm requires addition of data structure Q, which is shared by all LP's. Q contains a list of events along with the set of attributes required to evaluate termination condition C as well as simulation output measures.

<u>Modify S5 to read:</u>

```
Process event(s) with locally minimum timestamp;
Lock Q;
Insert each modified attribute set into Q;
Unlock Q;
```

<u>Add after S10:</u>

```
if(i=2){
  repeat
    for each attribute set in Q
      SV:=evaluate C(t);
  until (SV==true or empty(Q));
}
```

<u>Modify S12 to read</u>

Compute using "correct" values from Q;

Algorithms NST, NSQ and NPQ previously discussed cannot detect unstable termination conditions. To do this we need to evaluate the termination condition after every event occurrence which changes attributes in the domain of C. Implementation of this algorithm requires us to maintain a history of events and process attributes in the domain of C between barrier 1 and barrier 2. Algorithm EST does this by inserting the events into a shared queue in timestamp order as they are processed along with all the attributes required to compute C and simulation output measures. Between barriers 2 and 3, we evaluate C for each event in this queue in timestamp order using the values of the stored attributes. The algorithm terminates at the first evaluation of C yielding *true* and output measures are then calculated using the stored attribute values for that time, which are in the shared queue. In Section 5.1, we show that this algorithm imposes an overhead proportional to the number of events executed during the entire simulation.

4.5 Algorithm ESTR

<u>Modify S12 to read:</u>

```
Store time t at which termination was detected
Rollback to state at barrier 1 using stored attribute values
Re-execute until time t computing output measures
```

In Algorithm EST, we maintain in our history attributes necessary to determine termination as well as those necessary to compute output measures. Algorithm ESTR reduces the storage required by Algorithm EST by exploiting the fact that the attribute values needed to calculate output measures may in general be different from those required to determine termination. In the roll-back algorithm, the shared queue maintains only those attributes necessary to compute C, and we maintain the information necessary to compute output measures only at barrier 1. When the termination condition C becomes true, we store the time t at which termination was detected, roll-back to barrier 1 and re-execute the simulation until time t. We now have all the attributes required to calculate output measures. In comparison to algorithm EST, this algorithm requires less memory at the expense of having to perform some extra work after termination has been detected.

4.6 Algorithm EPT

This algorithm is a modification of Algorithm EST in which the evaluation of the termination condition C at each event occurrence in the shared queue is done using a combining tree.

4.7 Algorithm EPTR

This algorithm is a modification of Algorithm EPT with a roll-back mechanism to reduce the memory required.

We did not study the performance of algorithms EPT and EPTR because in our case the evaluation of the termination condition C took less time than the other simulation computations been done in parallel with it. However, they might be useful when C is a complicated function.

5 SYNCHRONOUS TERMINATION ALGORITHM PERFORMANCE

We report experiments on an N x N torus G/G/1 queueing networks. A 3 x 3 network is shown in Figure

5. Jobs which leave a server are randomly routed with equal probability on one of the N outgoing links. Initially, servers are idle and have the same number of jobs in their queues. We used an exponentially distributed service time with mean of 200 simulation time units and a constant of 100 time units added to it.

5.1 Predicted Performance

We can, on another plane, also classify termination conditions according to their dependence on the size of the problem being simulated. For example, consider a condition as follows: "stop when the total number of jobs processed by the system exceeds 100,000". It would take less simulation time for this condition to become true for a larger network than a smaller one simply because the larger network has a larger number of servers. On the other hand, consider a termination condition such as: "stop when the local virtual time of all *LP*'s first exceeds 100,000 units of simulation time". This termination condition would cause a simulation to stop at a simulation time which is independent of the size of the problem being simulated since all the *LP* would have to execute until that simulation time.

The termination conditions used in the experiments reported in Section 5.2 are the following:

T0: Algorithm NST & NSQ: "stop when the number of jobs processed exceeds 100,000 "

T1: Algorithm EST : "stop when the number of jobs processed equals 100,000 ".

T2: Algorithm NST, NSQ & EST: "stop when the local virtual time of each *LP* exceeds 100,000 ".

Let us define the following terms:

J : Total number of jobs processed by the system.

I : Number of iterations performed by the simulation algorithm.

T_N: Wall clock time taken by N LP's to perform one iteration of the simulation.

T_1: Wall clock time taken by one LP to perform one iteration of the simulation.

T : Wall clock time taken by the entire simulation, which is what we will be measuring.

Lubachevsky has shown that at least an order of N events are processed for every iteration of the algorithm (Lubachevsky 1989). Since the number of events is directly proportional to the number of jobs processed, we have:

Algorithm NST & NSQ with T0: The relationship $J / I = K * N$ holds for some constant K. Therefore $J = K * I * N$. Since J is constant (100,000) for T0, the product $N * I$ is constant.

So, $T = T_N * I = N * T_1 * I = A$ for some constant A, since T_1 is constant for Algorithms NST & NSQ, and $N * I$ is constant. Thus we should get a curve with slope 0 as N is varied.

Algorithm EST with T1: As before, $T = T_N * I = N * T_1 * I$. In this case T_1 is not constant but is proportional to N because the length of queue Q at the *LP* which computes the termination function is proportional to N.

Because the other nodes have to wait at barrier 3 for this node to evaluate the termination function, T_1 is proportional to N. So $T=B * N$ for some constant B and we should get a linear curve as we vary N.

Algorithm NST & NSQ with T2: For T2, since all the *LP*'s execute until the same simulation time, the number of iterations they perform is independent of N and is constant.

In this case $T = T_N * I = N * T_1 * I = D * N$, for some constant D, and we should get a linear curve as N is varied.

Algorithm EST with T2: In this case $T = T_N * I = N * T_1 * I = E * N * N$ for some constant E, since T_1 is proportional to N. Thus we should get a quadratic curve for this case.

5.2 Measured Performance

We study the performance of our algorithms using algebraic analysis and experimentation. We report on simulation experiments using a Sequent Symmetry shared-memory multi-processor with 10 80386 processors under the Dynix operating system. We used AT&T C++ 1.2.1 under the Presto thread package version 0.4 (Bershad, Lazowska and Levy 1987). We used Presto because it provides lightweight threads and several synchronization primitives for locks. We have the option of preemptive or non-preemptive thread scheduling with a user specified quantum size in the preemptive case. We report results using both these options.

We varied the size of the torus network being simulated from a 3 x 3 network to a 20 x 20 network for Algorithm NST, Algorithm NSQ and Algorithm EST. The number of processors is varied from 1 to 8. We chose 200 as the initial number of jobs for each server on all the networks simulated. We conjecture that this is a sufficient number of events to preclude the possibility of a server ever becoming idle. Each server is implemented by a thread executing the algorithm with its own copy of all local data. Barrier synchronization is effected using a master-slave mechanism based on examples distributed with Presto 0.4. The value of the bounded lag B is chosen so that each node examined only its nearest neighbors to calculate the earliest time another event could affect its history.

Figure 6 shows NST + T0 and EST + T1. We note that, as predicted the curve for algorithm NST+T0 is essentially flat, and the curve for algorithm EST+T1 is linear. There is an appreciable improvement in performance as the number of processors is increased form 1 to 2, and from 2 to 4. There is not much improvement if we use 8 processors instead of 4. This is because of the overhead introduced by the master-slave barrier mechanism in Presto.

The result of NST + T2 and EST + T2 is shown in Figure 7. As predicted, the graph for Algorithm NST+T2 is linear and the graph for algorithm EST+T2 is quadratic. As in the previous graph, we get performance improvements as we increase the number of processors from 1 to 2 and from 2 to 4, but not with 8.

The graphs for Algorithm NSQ+T0 are essentially the same as those for Algorithm NST+T0. Also , the graphs for NSQ+T2 are essentially the same as the graphs for NST+T2. We did not study the performance of EST+T0 as T0 is a stable termination condition and EST is redundant in this case.

We note that the results of our experiments are in accordance with the predicted values.

The data was collected while the simulation processes were the only user processes on the system. The variance in runtimes is less than 1 percent at the extremities, which are the cases for which the number of *LP*'s (N) equals 9 and 400. Therefore the data value used to plot the graphs is the mean of three experiments. The simulation was run both preemptively (with the quantum size varied between 100 msec and 600 msec) and non-preemptively, but this had no effect on the running times measured, perhaps because no event required more than 100 msecs of wall clock time to execute. For the one processor case, the simulation was run nonpreemptively without any locking of shared data structures because concurrent access is physically impossible in this case. The one processor case is, however, not a sequential simulation as there are multiple threads running on the single processor, which does impose an execution time overhead.

6 CONCLUSIONS

This paper presents seven termination algorithms for conservative-synchronous protocols. The first three detect stable termination conditions. Algorithm NST is a sequential evaluation algorithm, which evaluates the termination condition C after each event processing iteration. Algorithm NSQ is also a sequential evaluation algorithm, which evaluates C in parallel with event processing, and Algorithm NPQ is a parallel evaluation algorithm which parallelizes the evaluation of C using a combining tree. The other four algorithms detect both stable and unstable termination conditions. Algorithm EST does this by exhaustively evaluating C sequentially after each event occurrence. Algorithm ESTR does the same thing, but it does a roll back and re-execution after determining termination time t to calculate output measures and has less memory requirements than Algorithm EST. Algorithm EPT parallelizes the exhaustive evaluation of C in Algorithm EST using a combining tree, and Algorithm EPTR does the same with roll-back.

Our first observation is that the termination algorithms only required a few lines of code to be added to the basic Bounded Lag algorithm to include the termination condition. Our experiments with two terminating conditions using an N x N torus queueing network and the Bounded Lag simulation protocol reveal that detecting a stable terminating condition requires negligible overhead and detecting a nonstable termination condition requires $O(N)$ overhead.

There is an inherent attractiveness in using conservative-synchronous protocols with termination detection. First, the memory requirements are bounded by the size of the Bounded Lag. Second, it is easy to roll-back due to the synchrony, and this is exploited in algorithms ESTR and EPTR to reduce the storage required for termination and output measure reporting.We actually have two different ways to reduce memory overheads, embodied in Algorithm EST and Algorithm ESTR.

One avenue to explore would be to see the effect of increasing the Bounded Lag on our termination detection algorithms. This would increase the number of events processed between barriers. This should have a negligible effect on algorithms NST, NSQ and NPQ, which detect stable termination conditions,because they are independent of the number of events processed between the barriers. However, the remaining four algorithms, which detect non-stable termination conditions would be affected because their overhead grows with the number of events processed between barriers.

In comparison with optimistic protocols, with respect to rollback, conservative-synchronous protocols have less work to do to roll back as there because no anti-messages are requires and no cascading of rollbacks is possible. The optimistic protocol with a stable termination detection condition should require more time to evaluate a termination condition because each LP has a different local simulation time in general and the synchronization cost is high. However, this does not imply that the overall execution time of conservative-synchronous protocols is less than that of optimistic protocols.

ACKNOWLEDGEMENTS

Several discussions with Debra Richardson helped in formulating the conservative-synchronous termination algorithms presented in Section 4.

REFERENCES

Abrams, M. and D. Richardson .1991. Implementing a Global Termination Condition and Collecting Output Measures in Parallel Simulation. In *Proceedings of the 1991 Workshop on Parallel and Distributed Simulation* , 86-91. Anaheim, California.

Bershad, B. N., Lazowska, E. D. and Levy, H. M. 1987. "PRESTO: A System for Object-Oriented Parallel Programming". Technical Report, Department of Computer Science, University of Washington, Seattle.

Chandy, K. M. and J. Misra 1988. *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, Mass.

Chandy, K. M. and R. Sherman. 1989. Space-Time and Simulation. In *Proceedings of Distributed Simulation 1989, 53-59.* Tampa, Florida.

Lakshman, T.V.,Wei, V.K, On efficiently Computing Functions of Distributed Information.

Lubachevsky, B. 1989. Efficient distributed event-driven simulations of multiple loop networks. *Communications of the ACM* **32**: 111-123.

Mattern, F. 1987. Algorithms for Distributed Termination Detection. *Distributed Computing* **2**: 161-175.

Nicol, D. M. 1990. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. ICASE Report No. 90-20, NASA Langley Research Center, Hampton, Virginia.

Richardson, D. S. 1991. Terminating Parallel Discrete Event Simulations. Master's Thesis, Technical Report 91-9, Computer Science Department, Virginia Tech, Blacksburg, Virginia.

AUTHOR BIOGRAPHIES

VASANT SANJEEVAN is pursuing his M.S. in Computer Science at Virginia Polytechnic Institute and State University. He has a B.S. in Computer Science and Engineering from the Indian Institute of Technology, Delhi. His research interest is in Parallel Discrete-Event Simulation.

MARC ABRAMS : see the paper "SIMULATION PROGRAM DEVELOPMENT BY STEPWISE REFINEMENT IN UNITY" elsewhere in these Proceedings.

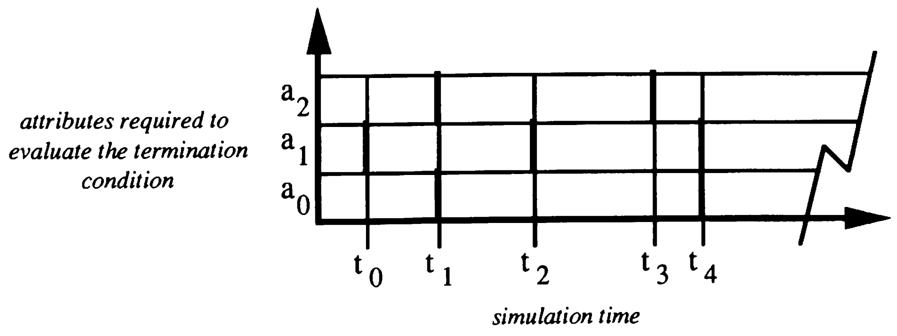

Figure 1: Illustration Of Space Time Rectangle For A Simulation With Three Attributes. The Heavy Vertical Lines Denote Assignment Of New Values To Attributes

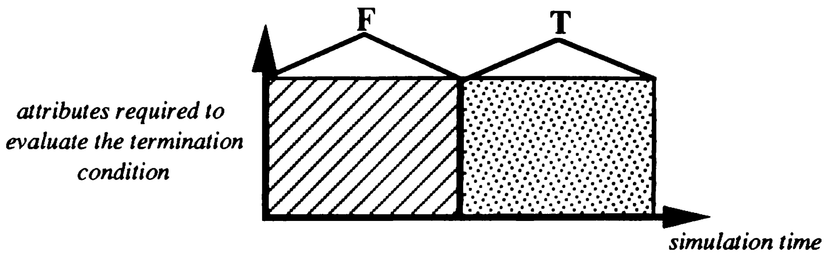

Figure 2: Example Of A Stable Termination Condition. "T" and "F" Denote Regions Of Time-Space In Which The Condition Is True And False, Respectively

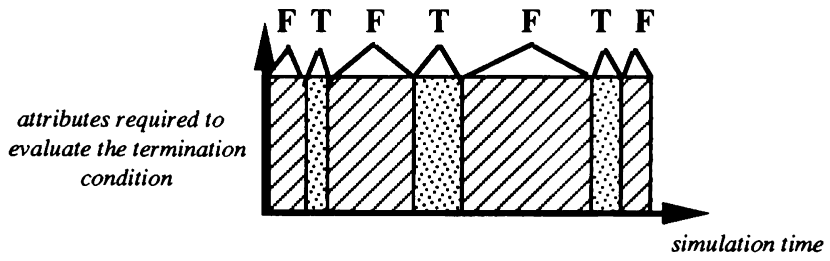

Figure 3: Example Of A Non-Stable Termination Condition

```
/*  Initially Floor:=0;SV:= false;   */
S0: compute T[i];
S1: while (not SV){
S2:    compute ALPHA[i];
S3:    synchronize;     /*  barrier 1 */
S4:    while(( T[i] < Floor + B) && (T[i] < ALPHA[i])){
S5:    process event(s) with locally minimum timestamp;
S6:    optionally schedule new events and/or delete events for
       LPi or other LP's;
S7:    delete the processed events from LPi;
S8:    recompute T[i];
       }
S9:    synchronize;    /*  barrier 2 */
S10:   if(i=1) Floor := min (T[i]), i = 1, N
S11:   synchronize;    /*  barrier 3 */
    }
S12:compute output measures
```

Figure 4: Algorithm For Logical Process i Of A Bounded Lag Simulation

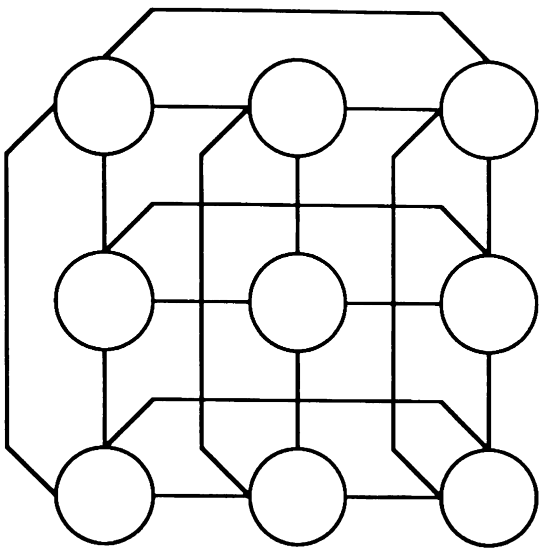

Figure 5: A 3 x 3 Torus Queueing Network

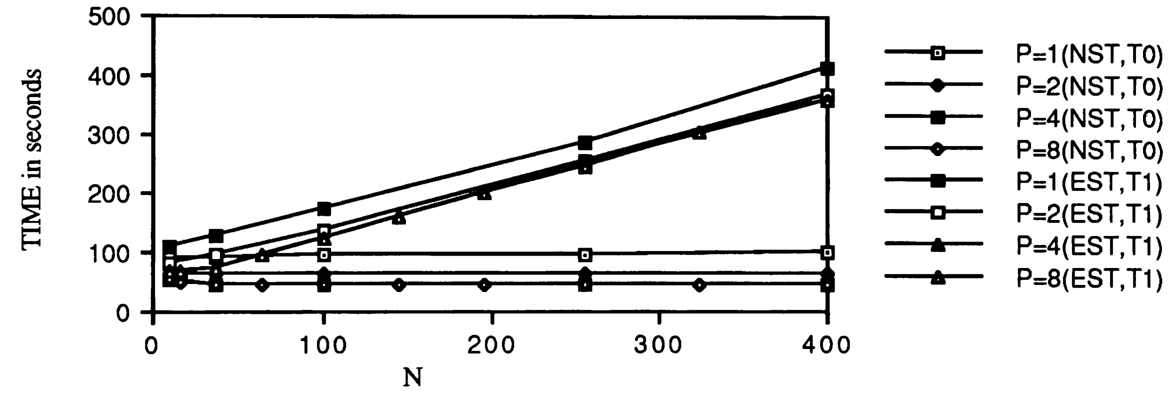

Figure 6: Wall Clock Time Required To Complete Simulation For N Threads With NST+T0 And EST+T1. P Is The Number Of Processors.

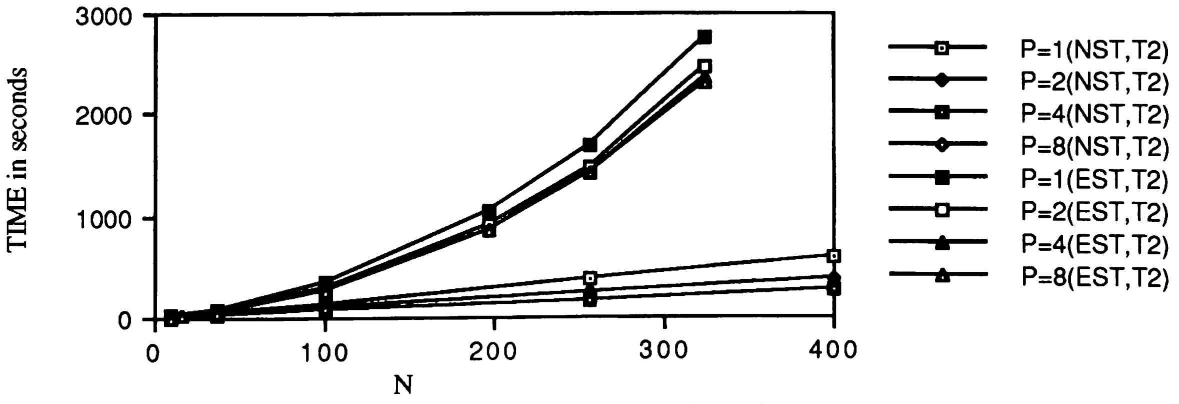

Figure 7: Wall Clock Time Required To Complete Simulation For N Threads With NST+T2 And EST+T2