1-1-2001

Lock-free Scheduling of Logical Processes in Parallel Simulation

Xiaowen Liu
Dartmouth College

David M. Nicol
Dartmouth College

King Tan Dartmouth College
false

Dartmouth Computer Science Technical Report TR2001-385

Lock-free Scheduling of Logical Processes in Parallel Simulation *

Xiaowen Liu, David M. Nicol, and King Tan

Department of Computer Science
Dartmouth College
Hanover, New Hampshire 03755
{*jasonliu,nicol,kytan*}*@cs.dartmouth.edu*

Abstract

With fixed lookahead information in a simulation model, the overhead of asynchronous conservative parallel simulation lies in the mechanism used for propagating time updates in order for logical processes to safely advance their local simulation clocks. Studies have shown that a good scheduling algorithm should preferentially schedule processes containing events on the critical path. This paper introduces a lock-free algorithm for scheduling logical processes in conservative parallel discrete-event simulation on shared-memory multiprocessor machines. The algorithm uses fetch&add operations that help avoid inefficiencies associated with using locks. The lock-free algorithm is robust. Experiments show that, compared with the scheduling algorithm using locks, the lock-free algorithm exhibits better performance when the number of logical processes assigned to each processor is small or when the workload becomes significant. In models with large number of logical processes, our algorithm shows only modest increase in execution time due to the overhead in the algorithm for extra bookkeeping.

1 Introduction

Parallel discrete-event simulation [2] has been proven to be a viable approach for simulations of a broad range of real-world applications, such as computer systems and communication networks. It is especially suitable for large-scale simulations that both require long execution time and consume vast memory space. However, the success didn't bring widespread acceptance of parallel simulation, which is still considered by the simulation mainstream as a niche research area. The complication of parallel simulation remains apparent in the user's manual of a parallel simulator. In many cases, the additional complexity in debugging and performance tuning does not fully justify the performance improvement brought by parallelism [10].

The difficulty of parallelizing discrete-event simulation is to preserve causality constraints among events. An event with smallest timestamp has the potential to change the state of the simulation and therefore

*This work is supported in part by DARPA Contract N66001-96-C-8530, NSF Grant ANI-98 08964, NSF Grant EIA-98-02068, and Dept. Of Justice Contract 2000-CX-K001

affect events that happen later in simulation time. This restrictive cause-and-effect relationship can be partially relaxed by partitioning the application into logical processes (LPs), where events on one LP can only be allowed to affect the state of that LP. State modification of other LPs can only be achieved by delivering events (sometimes called messages) to those LPs. Partitioning application into logical processes effectively separates the global event-list in a sequential simulation into multiple event-lists, one for each LP. The problem of parallelizing the simulation boils down to maintaining local causality constraints on each LP. That is, an LP only needs to decide whether to process the smallest event on its local event-list without introducing causality errors. Facing with the potential threat that other LPs would ship events with smaller timestamps to an LP, two major approaches separate parallel discrete-event simulation research: conservative and optimistic schemes. Conservative simulation [1] strictly prohibits out-of-order execution of events. The smallest local event cannot be processed unless it is guaranteed that there is absolutely no possibility having another event with smaller timestamp to arrive later at this LP. Optimistic simulation [5] allows events to be processed out of order. Once a causality error is detected, that is, when a "straggler" event with smaller timestamp later arrives at the LP, the simulation is rolled back to correct and recover from such error.

Our research focuses on parallel simulation on shared-memory multiprocessors in the context of simulation of large-scale communication networks. Since the memory utilization is considered essential in our application, we favor conservative simulation as it usually requires smaller memory footprint than most of its optimistic counterparts. It is well-known that lookahead in the simulation model is essential to the performance of conservative simulation. Lookahead is the ability to predict what will happen or will not happen in the simulated future. In loose terms, lookahead implies inherent asynchrony in the simulation model. With positive lookahead, an LP can process events and advance its local simulation clock well ahead of its neighbors. Extensive studies have been undertaken to illustrate the importance in extrapolating lookahead from the simulation model. In [9], Nicol gave a classification of lookahead based on different levels of knowledge extracted from simulation models. The use of different dimensions of lookahead underscores conservative synchronization protocols being developed so far. Poor lookahead results in more frequent synchronization since each synchronization only carries a little lookahead information. However, given that the lookahead is fixed in a simulation model, the overhead lies in the mechanism used for propagating the lookahead information in order for LPs to safely advance their local simulation clocks.

In this paper, we focuses on scheduling problems where many logical processes may be assigned to a processor. Our asynchronous algorithm extends the CMB algorithm [1] and applies to shared-memory multiprocessor machines, where, instead of sending NULL messages, each processor has an LP scheduler in charge of choosing the next LP ready for execution. It is well understood that the execution time of a conservative simulation is constrained by its critical path [4]. The optimal scheduling policy would choose the LP containing the event on the critical path. As pointed out by [11], it is however impractical to dynamically identify the event or logical process on the critical path. Yet several heuristics can be used to preferentially schedule LPs more likely on the critical path [14, 7]. In [11], the authors investigated the impact of various scheduling policies on the performance of conservative simulation.

To determine the safe time of an LP and ensure that an LP is scheduled as soon as it's ready, the scheduler needs to access variables potentially shared by different processors. A natural approach would be to use locks for mutual exclusion. In this traditional approach, shared objects are handled within critical sections to protect from simultaneous access by multiple processes. Mutual exclusion, however, has its disadvantage.

A slow process holding a lock may deter the access of fast processes and, therefore, form a bottleneck that degrades the performance of the entire system. Moreover, the cost associated with gaining access to locks may impose a considerable overhead to applications that are sensitive to low-level synchronizations. Inter-processor locks are expensive on shared memory multiprocessor machines. As in most cases, locks are implemented as busy-waiting. Not only it means a waste of CPU cycles, but also it becomes taxing on system traffic because of cache invalidations. Locks can also give rise to problems like priority inversion and deadlock. In fault tolerance studies, the use of locks is well-known to cause problems when a process in a critical section fails to relinquish the lock.

To avoid some of these problems, we use a lock-free asynchronous conservative parallel synchronization algorithm that is based on the *fetch&add* primitive to schedule logical processes. The algorithm is designed to avoid locks when calculating the safe time of an LP. The concept of lock-freedom, along with the concept of wait-freedom, comes from research in fault tolerant distributed computing. Informally, lock-free synchronization refers to the property that some operations on the shared object must complete in a finite number of steps. That is, no process can block the system of further operations to the shared object. It is still possible that starvation may happen, where some process fails to access the shared object, as it is constantly preempted by other faster processes. A stronger criterion called wait-free synchronization dictates that every process is guaranteed to complete operations in a finite number of steps. As we are more concerned with the efficiency of synchronization algorithms, wait-freedom is often too expensive to implement for synchronization. On the other hand, lock-free algorithms with careful implementation can be quite efficient [12].

Particularly, in our model of conservative parallel simulation, since the scheduler needs a snapshot of all incoming channels of an LP to calculate its safe time, the granularity of locks is larger than necessary and therefore may deter concurrency. Such problems become obvious when the number of logical processes per processor is small. Our lock-free algorithm avoids the problem and experiments show the algorithm has definitively better performance compared to the algorithm using locks. Moreover, since an LP may be waiting for more than one predecessor LP to advance its simulation clock, to determine which of the predecessor LPs is really responsible for scheduling the LP, a naive approach would involve unnecessary context switches by allowing all those predecessor LPs to take turns scheduling the LP. The lock-free algorithm can effectively avoid unnecessary context switches when dealing with critical channels.

The lock-free algorithm does not address all problems that come with using locks. We note that the use of *fetch&add* primitive still relies on mutual exclusion, albeit supported by hardware. On some shared memory machines, *fetch&add* primitive is not supported by the machine instruction set and, instead, it can only be emulated using other primitives. It should be noted, however, that it uses locks in a much smaller granularity. The lock-free algorithm in this case involves critical sections that are only used to enforce the atomic actions specified by the primitive. On the other hand, other algorithms with locks may have critical sections as large as the entire event processing session.

The structure of the paper is as follows. Section 2 defines the parallel discrete-event simulation model and some of the basic terms we will use to describe the algorithm. Section 3 presents related work and underlines the contribution of our approach. The lock-free scheduling algorithm is described in detail in section 4 and its correctness proof is given in section 5. Experiments and results are shown in section 6. Finally, section 7 gives our conclusions.

2 Model

The parallel discrete event simulation model is composed of a set of logical processes (i.e. LPs) connected by unidirectional channels through which messages are sent from sender LPs to receiver LPs. We assume that the topology of the graph does not change during simulation. Such a model can be represented as a directed graph, where the nodes in the graph represent the LPs and the edges represent channels that connect the corresponding LPs. Let t_i be the local simulation clock of LP_i. Define $\delta_{i,j} \geq 0$ to be the delay associated with the channel from LP_i to LP_j. Note that $\delta_{i,j}$ is a simple form of lookahead. The event sent from LP_i to LP_j at time τ will show up at time $\tau + \delta_{i,j}$ in LP_j's event-list. To avoid deadlock, we assume that the sum of the delays of edges along any cycle in the graph is strictly larger than zero. For LP_i, let $PRED(i)$ be the set of predecessor LPs of LP_i and let $SUCC(i)$ be the set of successor LPs of LP_i.

Let $T_{i,j}$ be the *channel time* from LP_i to LP_j. The channel time may be constantly updated by the source LP of the channel (i.e. LP_i) whenever the local simulation clock of LP_i is updated: $T_{i,j} = t_i + \delta_{i,j}$. It is understood that channel times may not be updated instantaneously. Therefore, at any time during simulation, we are only sure that $T_{i,j} \leq t_i + \delta_{i,j}$. Let H_i be the *safe time* of LP_i, defined as the latest simulation time up to which events in LP_i can be safely processed without introducing the risk of causality errors. It is guaranteed that no events with timestamp smaller than H_i will later arrive at LP_i. Theoretically, the safe time of LP_i can be derived at any time from a global snapshot of the simulation: $H_i = \min_{j \in PRED(i)}\{t_j + \delta_{j,i}\}$, where t_j is the current simulation time of LP_j at the snapshot. In our algorithm, LP_i only looks at the times of its incoming channels. We define $H_i' = \min_{j \in PRED(i)}\{T_{j,i}\}$ and say that LP_i is *ready* if and only if $t_i < H_i'$. In practice, it is unnecessary and too expensive to obtain a snapshot in order to compute the safe time. Instead, the algorithm only needs a lower bound of the true safe time, which we call *event execution horizon* and is denoted as h_i. It is obtained by traversing the incoming channels of LP_i and storing the minimum of current channel time $T_{j,i}$ in h_i. It is understood that the predecessor LP_j may not update $T_{j,i}$ promptly. However, this will only cause a smaller h_i to be calculated, since we are sure that $T_{j,i}$ is non-decreasing.

An LP scheduling algorithm is described informally as follows. When a ready LP_i is dispatched for execution, the h_i that it computes is sure to be greater than its local simulation clock t_i. All events within LP_i with timestamps smaller than h_i are then processed in timestamp order. During the process, new events may be generated. Events local to LP_i are inserted directly into the LP's event-list. External events are queued up in the output channels and will be later withdrawn by the destination LPs. Once all safe events in LP_i have been executed, LP_i must be suspended from execution. It is obvious that the safe time of LP_i is determined by the predecessor LP with the smallest channel time. The safe time of LP_i will not increase unless the predecessor LP with smallest channel time advances its local simulation clock. Borrowing terms from the literature, we call that LP with smallest channel time the *critical parent* of LP_i. The channel between LP_i and its critical parent is called *critical channel*. Note that there could be more than one critical parent of an LP at the same time; they all possess the smallest channel time. To advance the safe time of an LP, all its critical parents must advance their simulation clocks. When an LP is suspended from further execution, the dispatcher should be able to dispatch another ready LP for execution. We call such transition a *context switch*. The execution of an LP between two consecutive context switches is called an *execution session* or a *session* in short. The simulation ends when the timestamps of all LPs reach the predefined simulation end time $\mathcal{H}_\mathcal{T} > 0$ or some other termination condition is satisfied.

3 Related Work

It is well-known that conservative parallel simulation cannot beat the critical path [4]. To make things worse, it is considered as impractical to identify and therefore schedule events on the critical path as it unfolds dynamically. However, many conservative scheduling algorithms take advantage of the notion and strive to schedule events that are more likely to be on the critical path.

Stepping up at a higher level, scheduling critical events can be approximated as scheduling critical LPs[1]. When a ready LP is scheduled for execution, the session continues until the local simulation clock reaches its safe time. The LP cannot continue until its critical parents advance the channel times. The problem then translates into an LP scheduling policy that can preferentially schedule LPs that contain critical events. Naroska and Schwiegelshohn's scheduling algorithm [7] gives priorities to logical processes that directly influence computations on other processors. The communication latency among processors with distributed memory is more likely to put events sent between them on the critical path. An empirical performance study of alternative LP scheduling policies is presented in [11]. The paper examined a number of heuristics that can be used to predict LPs on the critical path. They include scheduling LPs on the basis of LP's next event timestamp, safe time, or local simulation clock. While the paper showed that some heuristic is better than others, the authors pointed out that the benefit of the scheduling algorithms is sensitive to overheads, e.g. sorting cost.

In [14], the authors introduced a multi-level scheduling algorithm, called Critical Channel Traversing (CCT), that achieves efficiency in simulation with low granularity events. At highest level of scheduling, tasks (i.e. LP groups) are scheduled as a single entity from a central priority queue where tasks are placed in increasing timestamp order. Within a task, the scheduling algorithm preferentially chooses LPs with the largest number of events that are ready to process. The algorithm uses spin-locks to synchronize logical processes when accessing shared variables in channels.

In order to show comparison to our algorithm, the pseudo-code of the CCT algorithm that deals with LP scheduling is listed as follows for easy reference later. Note that the algorithm uses spin-lock at line 15. Later in the experiment section, we will compare performance of the CCT algorithm with our lock-free algorithm.

CCT LP Scheduling Algorithm

```
1:  WT_i := ∞
2:  foreach incoming channel (j, i)
3:      busy_{j,i} := 1
4:      critical_{j,i} := 0
5:      if T_{j,i} < WT_i then
6:          WT_i := T_{j,i}
7:          Keep_c_id_i := i
8:  Process safe events in order
9:  t_i := WT_i
```

[1]The notion of execution session of an LP does not necessarily mean a session should always run to completion, that is, until the local simulation clock reaches its safe time. The next critical event might not be on the same LP. Here, the approximation suggests that LP context switches in most cases are more costly and therefore should be avoided whenever possible. Of course, critical path may not be determined dynamically in an obvious way.

10: $critical_{Keep\_c\_id_i,i} := 1$
11: **foreach** incoming channel (j, i)
12: $busy_{j,i} := 0$
13: **foreach** outgoing channel (i, j)
14: $T_{i,j} := t_i + \delta_{i,j}$
15: **repeat until** $busy_{i,j} = 0$
16: **if** $critical_{i,j}$ **then** unblock LP_j

It is worthwhile to point out that LP scheduling with execution session running as long as possible inherently admits that the overhead caused by context switching is non-trivial. Depending on the application, longer execution session (i.e. fewer context switches) does not necessarily mean more savings in synchronization of parallel simulation. However, from scheduling point of view, fewer context switches not only translates to better memory locality and higher cache utilization, but also it means less cost for sorting LPs in the ready queue. On the other hand, the real time delay associated with updating subsequent channel times may give rise to artificial blocking [13], where the successor LP is waiting for the channel time to be updated while the predecessor LP has advanced its local simulation clock. There is a tradeoff for the length of an execution session before the outgoing channel times should be updated. It hinges on the scheduling overhead. Our algorithm aims to reduce the overhead when scheduling logical processes without the use of mutual exclusive locks. As a design decision, the number of context switches can be also reduced by averting unnecessary context switches caused by the existence of multiple critical parents. The algorithm uses *fetch&add* primitive, which is explained next.

4 The Algorithm

4.1 The *fetch&add* operation

Our scheduling algorithm uses the *fetch&add* primitive. A *fetch&add* operation, *fetch&add*(loc, k) where *loc* is a memory location and k is an integer, is defined as follows: Let m be the value of *loc* immediately before the *fetch&add* operation. Then m is the value returned by the operation and $m + k$ is the new value of *loc*. While some shared-memory machines directly provide *fetch&add* primitive as a well-defined machine instruction, the operation can be easily implemented using other intrinsics. For example, the operation can be implemented using a sequence of load-linked and store-conditional instructions in a loop.

4.2 Intuition

We now describe the intuition behind our algorithm. To achieve lock-free synchronization, we use tokens as a means of communication among LPs. When LP_i detects that its predecessor LP_k is a critical parent, LP_i *issues* a token for LP_k. When LP_k next advances the channel time $T_{k,i}$ (thus ensuring that LP_k is no longer a critical parent of LP_i), LP_k will *retrieve* the token. LP_k must then *return* the token to LP_i, thereby notifying LP_i that LP_k is no longer a critical parent. Since LP_i may have $u (u \geq 1)$ critical parents, LP_i is ready (i.e. $t_i < H_i'$) if and only if all u tokens issued by LP_i are returned. A counter CNT_i records the number of unreturned tokens issued by LP_i. When $CNT_i = 0$, then LP_i is ready, and should be unblocked and made available for dispatch. The description above is intentionally imprecise; the behavior of the algorithm

is considerably more complex. We provide rigorous claims about our algorithm in the lemmas and theorems to follow.

We now describe the implementation of the tokens and CNT_i. When LP_i issues a token for LP_k, LP_i writes 0 to $TOK_{k,i}$. Both LP_k and LP_i may attempt to retrieve a token previously issued by LP_i. An LP attempts to retrieve a token from $TOK_{k,i}$ by executing *fetch&add*($TOK_{k,i}$, 1). If the value returned by this operation is 0, then the token is retrieved. Otherwise, no token has been retrieved. After a token has been issued and before the next token is issued, the first attempt to retrieve a token will succeed, and all subsequent attempts will fail. CNT_i is implemented as follows: LP_i periodically adds the number of recently issued tokens that LP_i expects to be returned, say x, to CNT_i by executing *fetch&add*(CNT_i, x). A predecessor of LP_i returns a token to LP_i by executing *fetch&add*(CNT_i, -1). If the new value of CNT_i after executing any of these *fetch&add* operations is 0, then all tokens issued by LP_i have been returned.

4.3 Description of the algorithm

LOCK-FREE SCHEDULING ALGORITHM

```
1:  N := 0
2:  Compute h_i := min_{k∈PRED(i)}{T_{k,i}}
3:  if t_i = h_i then goto 12
4:  Execute safe events in order
5:  t_i := h_i
6:  foreach LP_k ∈ SUCC(i)
7:      T_{i,k} := t_i + δ_{i,k}
8:      x := fetch&add(TOK_{i,k}, 1)
9:      if x = 0 then
10:         y := fetch&add(CNT_k, -1)
11:         if y = 1 then unblock LP_k
12: m := 0
13: foreach LP_k ∈ PRED(i)
14:     if T_{k,i} = h_i then
15:         TOK_{k,i} := 0
16:         if T_{k,i} = h_i then
17:             N++
18:             m++
19:         else
20:             x := fetch&add(TOK_{k,i}, 1)
21:             if x ≠ 0 then N++
22: if m = 0 then goto 2
23: else
24:     x := fetch&add(CNT_i, N)
25:     if x = -N then goto 1
```

Our algorithm is executed by any LP_i when it is dispatched to start a new session. It makes no as-

sumption about how the dispatcher selects an LP for execution from the pool of unblocked LPs. During the initialization of the simulation, each channel time $T_{i,j}$ is set to $\delta_{i,j}$. For each LP_i, the local simulation clock t_i is set to zero and the safe time is computed as $H_i = \min_{j \in PRED(i)}\{\delta_{j,i}\}$. If $H_i > 0$, then LP_i is entered into the pool of unblocked LPs available for dispatch. If $H_i = 0$, then LP_i is blocked. For all $j \in PRED(i)$, $TOK_{j,i}$ is set to zero if $\delta_{j,i} = 0$. Otherwise, it is set to one. CNT_i is set to the number of zero-delay predecessor channels.

We now describe our algorithm as executed by LP_i. N is used to temporarily store the number of tokens issued by LP_i before it is added to CNT_i. At the beginning of a session, N is initialized to 0. LP_i then computes h_i, the event execution horizon (line 2). During a session, line 2 may be executed more than once. Each execution of line 2 initiates a new *round*. Rounds are numbered consecutively $1, 2, \cdots$, regardless of whether a new session has started. A round in which $t_i = h_i$ at line 3 is called *futile at time* t_i. Otherwise it is called *non-futile*. If $t_i = h_i$ at line 3, then there are no safe events to be executed in the current round, and no updates to LP_i's successor channel times are necessary. Therefore, LP_i bypasses lines 4-11 that process events and updates the successor channel times. If $h_i > t_i$, safe events are executed in order, and t_i is advanced (lines 4,5). For each of its successor channels, LP_i updates the channel time $T_{i,k}$ (line 7) and attempts to retrieve a token (line 8). If a token is indeed retrieved (this event is called *token retrieval at channel time* $T_{i,k}$), it is returned to the token issuer LP_k by decrementing CNT_k by one (line 10). If the execution of line 10 by LP_i results in $CNT_k = 0$, LP_i signals the unblocking of the next available blocked LP_k. The execution of line 10 that results in $CNT_k = 0$ is called an *unblock* event. LP_i next scans its predecessor channel times to see if any such channel time $T_{k,i}$ equals h_i. If $T_{k,i} = h_i$ at both lines 14 and 16, then LP_k is a *critical parent* of LP_i. Otherwise LP_k is *non-critical*. If $T_{k,i} = h_i$ at line 14, LP_i issues a token for LP_k (line 15). LP_i then reads $T_{k,i}$ again. If $T_{k,i} = h_i$ at line 16, then the token issued at line 15 is considered *delivered* (to a critical parent) and N, an interim counter of the number of delivered tokens, is incremented accordingly. m is also incremented. A positive m reflects the presence of a critical parent. However, if $T_{k,i} = h_i$ at line 14 and $T_{k,i} > h_i$ at line 16 (LP_K is non-critical), then LP_i attempts to retrieve the token that it issued at line 15. If LP_i successfully retrieves the token, then the token is not delivered. Otherwise it is considered *delivered* (to a non-critical parent). N is incremented (line 21) to reflect token delivery to a non-critical parent. We observe that a token delivered to a non-critical parent LP_k is known to have been retrieved by LP_k and will eventually be returned (i.e. CNT_i will eventually be decremented by LP_k).

At line 22, $m \neq 0$ if and only if there is a critical parent. If $m = 0$ at line 22, we call the current round *green*. When the current round is green, LP_i proceeds to the next round, starting at line 2 (hence not changing N). t_i is certain to advance in a round following a green round. If $m > 0$ at line 22, we call the current round *red*. In this case, LP_i adds N to CNT_i. If the resulting $CNT_i = 0$, then LP_i proceeds to the next round, starting at line 1 (thus setting N to 0). Such a red round is called *red-go*. If the resulting $CNT_i \neq 0$, then LP_i becomes blocked. Such a red round is called *red-stop*. The execution of line 24 is called a *boost* event. A boost event occurs only during a red round. A boost event is called *red-go* (*red-stop*) if it occurs during a red-go (red-stop) round. A blocked LP becomes unblocked (available to be dispatched to execute a new session) whenever there is an unexpended unblock signal (generated by an unblock event). (An unblock signal is expended if it has been used to unblock an LP.)

4.4 Empirical Considerations

As we mentioned earlier, the real time delay associated with updating subsequent channel times may give rise to artificial blocking. The tradeoff between reducing the number of context switches and increasing channel updates is balanced on the total overhead and is beyond the scope of this paper. In Section 4.3, for the sake of easy exposition, we showed that the algorithm processes all safe events up to the measured event horizon (at line 4) before updating subsequent channel times and then unblocking successor LPs (from line 6 to 11). In practice, the two steps can be interwoven.

Source LPs have no incoming channels and their event horizon will be infinite. Without proper flow control mechanism, they will run to completion in a single session. Sometimes this will create problems as generated events could exhaust memory. In our implementation, two sets of flow control mechanisms are used. The user can define a time period in which source LPs must reschedule themselves. Or, the user can set a memory threshold value for each execution session. When the generated events cause the memory usage above the threshold, LPs must reschedule themselves. A combination of both schemes can be also used. Flow control is extremely important especially when memory is tight for large-scale simulations. By no means our schemes reached total satisfaction. It warrants further study.

A problem with real-world *fetch&add* primitive is that it may wrap around since *fetch&add* operates on integer values. This is not a major concern in practice. Nonetheless, the problem can be detrimental if it happens in extreme cases. Here we present a simple solution for our algorithm. To obtain token (at line 8 and 20), we can instead use atomic swap to exchange values between TOK and one. Atomic swap is either provided as an machine instruction or can be easily implemented using other primitives. The range of values for CNT_i is dictated by the variable N. An additional condition can be added to line 22 so that when N is greater than certain threshold, the control will branch to line 24 and bring CNT_i back to normal range.

It should be noted that access to channel time $T_{i,j}$ must be sequentially consistent. This could be a problem if the simulator chooses to use larger or complicated data types whose read and write are not guaranteed sequentially consistent [8]. A vast literature exists for constructing atomic registers in distributed computing. In our implementation, for the case of large timestamp types, we use an algorithm that requires very little additional memory [3]. Note that even sequential consistency is too strong on some of the shared memory machines. In that case, special care must be taken to read and write operations. Memory barrier instructions ought to be used to guarantee sequential consistency.

The algorithm does not specify how those newly generated events in the execution session are delivered to their destinations. It is simple to insert local events directly into the local event-list. For remote events, they can be queued up in the channels, from which they will be withdrawn when the destination LPs reach their execution sessions. Synchronization of the access is needed. Also, the algorithm does not specify the mechanism to unblock subsequent LPs. The unblocked LP could get into the ready list of the current processor. In our implementation, we prefer making the LP stay at its original processor for the sake of locality. In that case, further synchronization is needed.

5 Correctness Proof

In this section, we outline a proof of correctness of the algorithm. Since the algorithm does not deviate from existing LP scheduling algorithms, we omit the proof that events are processed in timestamp order in LPs (i.e. local causality constraint). However, we need to show that the algorithm does terminate. Consider

that the simulation starts at time zero and terminates when all LPs have reached a predefined simulation end time $\mathcal{H}_\mathcal{T} > 0$. We need to prove that all LPs will reach time $\mathcal{H}_\mathcal{T}$. We make two standard assumptions in the following proofs:

A1: The sum of the channel delays along any cycle is positive.

A2: The events of an LP within a finite time interval can be executed in finite time.

Lemma 1 *After a boost event (the execution of line 24) in round r, and before the beginning of round $r+1$, CNT_i = (number of tokens delivered by LP_i)− (number of returned tokens) ≥ 0 and its value is non-increasing during the interval in question.*

Proof: N cumulates the number of delivered tokens (lines 17,21) until a boost event (in round s, say) when N is added to CNT_i. N is then reset to 0 at the beginning of round $s+1$. (s and $s+1$ may or may not belong to the same session.) Hence, during the interval in question (after the boost event in round r, and before the beginning of round $r+1$), the number of tokens delivered by LP_i (in all rounds through r) has been added to CNT_i. Further, each returned token has been accompanied by an execution of line 10 that decrements CNT_i by one. Therefore, CNT_i = (number of tokens delivered by LP_i)− (number of returned tokens) during the interval in question. Since each returned token has been previously delivered, $CNT_i \geq 0$. During the interval in question, CNT_i may only be changed by an execution of line 10, and not by a boost event, hence its value is non-increasing.

Lemma 2 *Let r be a red round of LP_i. At the beginning of round $r+1$, $CNT_i = 0$ and all tokens delivered by LP_i have been returned.*

Proof: If r is a red-go round, the proof follows from Lemma 1, since a red-go boost leaves $CNT_i = 0$. Now suppose r is a red-stop round. We observe that the value of CNT_i is increased by boost events (line 24). Red-stop boost leaves $CNT_i > 0$. Red-go boost leaves $CNT_i = 0$. The value of CNT_i may decrease or remain unchanged, but not increase, between two boost events. Recall that an unblock event is an execution of line 10 by a predecessor of LP_i that results in $CNT_i = 0$. Therefore there is at most one unblock event between a red-stop boost and the next boost. Also, there is no unblock event between a red-go boost and the next boost. Thus, at any time, (number of red-stop boosts) $\geq$ (number of unblock events).

At the beginning of round $r+1$, each of the past red-stop rounds of LP_i can be paired with an unblock signal (generated by an unblock event) that was expended on its behalf (since there is a next round that follows each of these red-stop rounds). Hence, (number of red-stop boosts) $\leq$ (number of unblock events). Therefore, (number of red-stop boosts) = (number of unblock events). This means that there is an unblock event that follows the red-stop boost in round r. By Lemma 1, this implies that at the beginning of round $r+1$, $CNT_i = 0$, and all tokens delivered by LP_i have been returned.

Theorem 5.1 *Let LP_i be ready and blocked, with $t_i \leq \mathcal{H}_\mathcal{T}$. Then LP_i is eventually unblocked.*

Proof: Let the last round LP_i executed before LP_i became blocked be round s. (Round s is therefore a red-stop round.) Let $r (r < s)$ be such that round r is red, rounds $r+1, r+2, \cdots, s-1$ are green. By Lemma 2, at the beginning of round $r+1$, all tokens delivered by LP_i in rounds $1, 2, \cdots, r$ have been returned. Therefore, after the red-stop boost in round s, CNT_i = (number of unreturned tokens delivered

in rounds $r+1, r+2, \cdots, s$). We observe that: (1) All tokens delivered in a green round (hence, in rounds $r+1, r+2, \cdots, s-1$) are delivered to non-critical parents. (2) Any token delivered to a non-critical parent has already been retrieved by the parent (by the time LP_i executes line 20), and will eventually be returned. These observations imply that if all tokens delivered to critical parents in round s are eventually returned, then eventually $CNT_i = 0$.

By assumption, LP_i is ready. Thus, each critical parent LP_k has already updated channel time $T_{k,i}$ to some value $\tau_{k,i} (\tau_{k,i} > t_i)$. The attempt by LP_k to retrieve a token at channel time $\tau_{k,i}$ happens after LP_i issues a token for LP_k (at line 15) in round s. If the token has not been previously retrieved by LP_k, it certainly will be retrieved by the attempt of LP_k at channel time $\tau_{k,i}$. Therefore all tokens delivered to critical parents in round s are eventually returned. This means that eventually $CNT_i = 0$ and that there is an unblock event that causes LP_i to be unblocked.

Lemma 3 *There is some $\Delta > 0$ such that, for all LP_i, the value of t_i increases by at least Δ in every non-futile round.*

Proof: The proof is similar to the proof of boundedness property in [1]. Here we give a sketch of our proof. Any t_k value is a linear combination, with non-negative coefficients, of the channel delays $\delta_{i,j}$. Thus, any t_k value can be written as $\sum_{i,j} c_{i,j}\delta_{i,j}$ where each $c_{i,j} \geq 0$ is an integer. Let $\delta_{min} = min_{i,j}\{\delta_{i,j}\}$. Since we deal only with t_k values not exceeding $2\mathcal{H}_\mathcal{T}$, each $c_{i,j} \leq \lceil 2\mathcal{H}_\mathcal{T}/\delta_{min} \rceil$.

Let sets $S = \{\sum_{i,j} c_{i,j}\delta_{i,j} | 0 \leq c_{i,j} \leq \lceil 2\mathcal{H}_\mathcal{T}/\delta_{min} \rceil, c_{i,j} \text{ an integer}\}$, $T = \{x - y | x > y; x, y \in S\}$. Clearly S is finite. Hence T is finite too. Let $\Delta = min\,\{x | x \in T\}$. Since $x \in T$ implies $x > 0$, and T is finite, we have $\Delta > 0$. Let a, b be two successive values of t_k such that $a < b$ and both $a, b \leq 2\mathcal{H}_\mathcal{T}$. Since $a, b \in S, b - a \in T$. Therefore $b - a \geq \Delta > 0$.

Lemma 4 *For any time τ, there is at most one round of LP_i that is futile at τ.*

Proof: Let round r of LP_i be futile at τ. Round $r-1$ must be red. There must be a predecessor of LP_i, say LP_k, that is critical with channel time $T_{k,i} = \tau$ in both rounds $r-1, r$. We claim that event (E1) precedes (E2), which in turn precedes (E3), where:

E1: LP_i executes line 14 in round $r-1$, reading $T_{k,i} = \tau$.

E2: LP_k retrieves the token delivered in round $r-1$ by LP_i.

E3: LP_i executes line 2 in round r, reading $T_{k,i} = \tau$.

(E1) precedes (E2) because (E1) precedes LP_i issuing the token for LP_k at line 15 of round $r-1$, which in turn precedes (E2). (E2) precedes (E3) because, by Lemma 2, the token delivered for LP_k in round $r-1$ has been returned at the beginning of round r.

Let α be such that event (E2) is LP_k's retrieval of token at channel time $T_{k,i} = \alpha$. If $\alpha < \tau$, then at (E1), LP_i must read $T_{k,i} < \tau$ (because (E1) precedes (E2)). If $\alpha > \tau$, then at (E3), LP_i must read $T_{k,i} > \tau$ (because (E3) follows (E2)). Therefore, $\alpha = \tau$. We have shown that if round r is futile at τ, then there is a critical parent LP_k that retrieves a token delivered by LP_i (in round $r-1$) at channel time τ.

Suppose there are more than one futile rounds at τ. Without loss of generality, let rounds $r, r+1$ be futile at τ. By the same argument, there is a critical parent LP_k who retrieves at channel time $T_{k,i} = \tau$ both the token delivered in round $r-1$ and the token delivered in round r. However, from line 3, there is

at most one token retrieval by LP_k at channel time τ. This contradiction proves that there is at most one round of LP_i that is futile at τ.

Lemma 5 *Any unblocked LP eventually gets dispatched and executes a new session.*

Proof: We claim that the total number of rounds of all LPs is finite. Since the number of LPs is finite, it suffices to show that the total number of rounds of any LP, say LP_i, is finite. By Lemma 3, $t_i > \mathcal{H}_\mathcal{T}$ after LP_i executes a certain finite number of non-futile rounds. Hence, the number of non-futile LP_i rounds is finite. By Lemma 4, there is at most one futile round at each t_i value. Thus, the total number of futile rounds of LP_i is also finite. Therefore the total number of LP_i rounds is finite.

Assumption (A2) (events within a finite interval can be executed in finite time) implies that each round completes. Since the total number of rounds is finite, each session completes. The fact that the total number of rounds is finite also implies that the total number, over the duration of an entire simulation, of unblocked LPs entering the pool of LPs awaiting dispatch is finite. This, together with the fact that each session completes, implies our Lemma.

Theorem 5.2 *Eventually all* $t_i > \mathcal{H}_\mathcal{T}$.

Proof: As shown in the proof of Lemma 5, the number of rounds of LP_i is finite and each round completes. Therefore there exists a time when no more rounds are initiated or executed, all t_i and all channel times $T_{i,j}$ are frozen forever (with no further changes to their values). Consider such a time. Assume, by contradiction, $t_{i_0} \leq \mathcal{H}_\mathcal{T}$ at such a time. LP_{i_0} is either forever blocked, or forever in the unblocked LP pool awaiting dispatch. By Lemma 5, LP_{i_0} cannot be forever in the unblocked LP pool. Thus, LP_{i_0} is forever blocked. This implies that LP_{i_0} must be forever not ready. (If LP_{i_0} is eventually ready, then, by Theorem 5.1, it is eventually unblocked. This is a contradiction.) Let LP_{i_1} be a critical parent of LP_{i_0}. Then, $t_{i_1} = t_{i_0} - \delta_{i_1,i_0} \leq t_{i_0} \leq \mathcal{H}_\mathcal{T}$. This implies that LP_{i_1} is forever not ready (by the same argument used for LP_{i_0}). In general, let $m = 1, 2, \cdots$. Let LP_{i_m} be a critical parent of $LP_{i_{m-1}}$. Then $t_{i_m} = t_{i_{m-1}} - \delta_{i_m,i_{m-1}} \leq t_{i_{m-1}} \leq \mathcal{H}_\mathcal{T}$ and LP_{i_m} is forever not ready. By assumption (A1) (the sum of the channel delays along any cycle is positive), $LP_{i_0}, LP_{i_1}, \cdots$ is an infinite chain of *distinct* LPs. This is however impossible since the total number of LPs is finite. This completes our proof by contradiction.

6 Experiments

We implemented both the critical channel scheduling part of the CCT algorithm and the lock-free algorithm. The algorithms are embedded as part of the Dartmouth Scalable Simulation Framework (DaSSF) kernel [6]. **SSF** is a process-oriented, conservatively synchronized parallel simulator, designed for but not exclusively for simulating large scale multi-protocol communication networks. As part of the synchronization mechanism, the scheduling algorithms work with process dispatching unit in the simulator where ready LPs are queued for execution according to the timestamps of the earliest events.

All our experiments were conducted on a Sun Enterprise 6500 that has 14 400MHz UltraSPARC-II processors with a total of 7 GB memory. Each processor is equipped with 8 MB secondary external cache. The system chassis consists of 7 CPU/Memory boards interconnected by Gigaplane system bus that can deliver up to 2.88 GB/sec peak rate data transfer. The machine is running Solaris 7 operating system.

Every data collected in the experiments is an average value from five simulation runs. Also, almost all samples, the 95% confidence interval is less than 5% around the sample mean[2].

The benchmark model for our experiments contains an array of interconnected logical processes (called *entities* in **SSF**). The simple model is chosen because we only want to exercise synchronization overhead. Also, we do not allow messages to be sent between LPs and we do not want more than one event in the local event-list. Delivering external events or queueing more than one event will exercise other parts of the simulator that we are not interested in this study.

Several parameters control the instantiation and deployment of the model onto the multiprocessor machine. The parameter N specifies the number of LPs in the model. These LPs are logically arranged as a ring. The connectivity of the LPs is specified by the parameter R, which is called *connection radius*. Each LP connects to its neighbors within a radius of R. That is, every LP connects to R LPs ahead of it and R LPs behind it in the ring structure, effectively making all connections bidirectional. Each channel has a constant delay of one unit simulation time. These N LPs will be deployed evenly among P processors upon execution: LP_i will be statically assigned to processor ($i \ mod \ P$). Each LP will have at most one local event in its event-list. Upon processing the event, another local event will be scheduled with an exponentially distributed increment into the future. The length of the interval is controlled by the parameter D, called *event density*. An event density of D means there is an average of D events to be processed per unit simulation time. Note that if D is zero, no event is processed, which means that there is no workload and the simulation engine only relies on the synchronization mechanism to schedule LPs in order to advance their local simulation clocks.

In the first experiment, we set the number of LPs in the model to be the same as the number of processors used to run the bench mark application; each processor has only one LP assigned to it. In all cases, the LP graphs in the model are fully connected, that is, $R = \lfloor N/2 \rfloor$. We varied the number of processors and the event density. Figure 1 shows the ratio of the execution time of our lock-free algorithm to that of the CCT algorithm. This experiment demonstrates relative synchronization effect of the two scheduling algorithms. It is clear from the figure that, as event density increases, the lock-free algorithm performs much better than CCT. The result is expected, since, in the CCT algorithm, every LP in the model tends to block all other LPs during its execution session. This justifies our concern about algorithms using locks in which the delay of a single process in a critical section forms a bottleneck of the entire system. With moderate to heavy workload in the model, this experiment actually represents the worst case scenario for the CCT algorithm, since all LPs are fully connected, that maximizes the blocking effect, and there is only one LP per processor, that effectively avoids the benefit of latency hiding that can be achieved by assigning multiple LPs to each processor. As the result, we observed savings in execution time by the lock-free algorithm as much as 48%. However, it must be pointed out that, in the case of no workload (i.e. D=0), the lock-free algorithm also experiences slowdown, especially when the number of processors is small. We conjecture that this is due to the extra complexity in the lock-free algorithm for manipulating variables used for bookkeeping.

Figure 2 shows the ratio of the number of context switches of the lock-free algorithm to that of the CCT algorithm. As expected, the lock-free algorithm reduces the number of context switches. The saving becomes more noticeable as the event density increases, that is, when the workload becomes significant. Close inspection on the raw data showed that, varying event density, the lock-free algorithm has roughly constant number of context switches, while the number increases for the CCT algorithm. We noticed that

[2]Only a few data points have larger variations that required more than five runs.

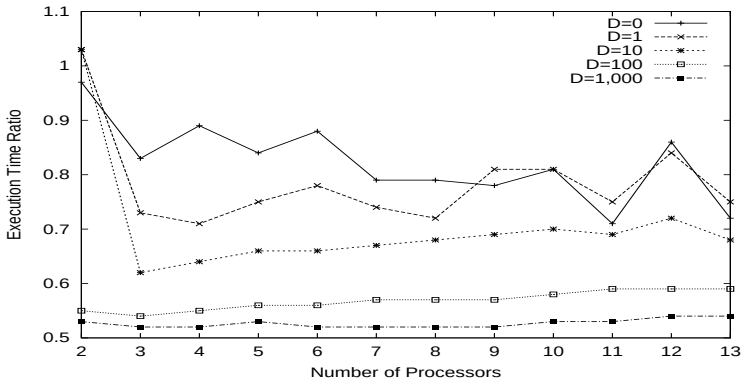

Figure 1: The ratio of execution time of the lock-free algorithm to that of the CCT algorithm in the worst-case scenario of CCT.

as the workload increases, the difference between fast and slow processors becomes obvious. Since it is more likely that LPs are each running at a different pace, there is a smaller chance of an LP having its critical channels updated when there are more than one critical parents present. These unnecessary context switches are less likely to be avoided by CCT and therefore result in smaller context switch ratios for the lock-free counterpart.

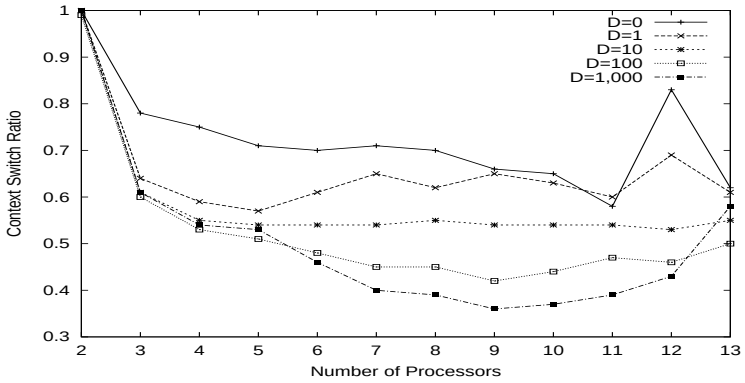

Figure 2: The ratio of the number of context switches of the lock-free algorithm to the CCT algorithm in the worst-case scenario of CCT.

In the second experiment, we increased the number of logical processes assigned to each processor. As more LPs are ready for execution, blocking is less of a system bottleneck. In this experiment, we fixed the number of processors at 12 and increased the number of LPs assigned to each of the processors. The experiment was conducted in three settings with different event densities (0, 100, and 10,000) to represent different levels of workload. In each setting, we also changed the connection radius: one with connection radius being one, one with the radius being six, and one with fully connected LP graphs (denoted by infinite radius). The result is shown in Figure 3. The curves represent changing execution time ratio of our algorithm to the CCT algorithm. Note that the x-axes are in logarithmic scale.

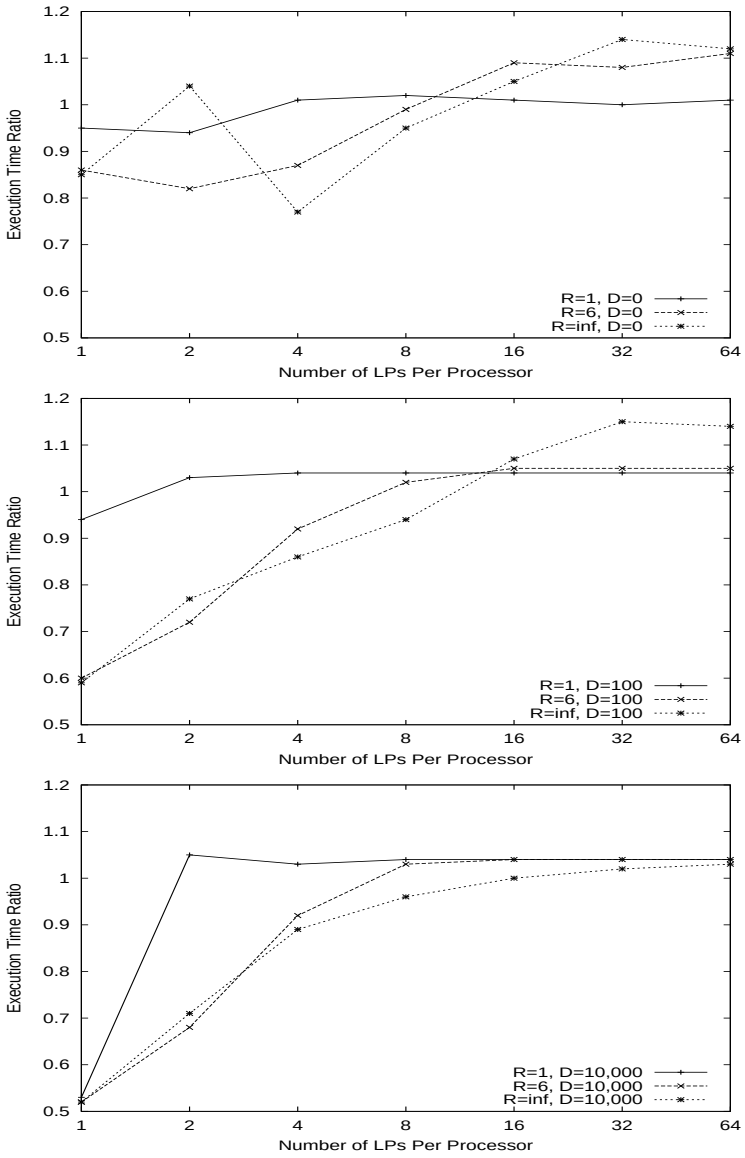

Figure 3: The ratio of the execution time of the lock-free algorithm to that of the CCT algorithm with increasing number of logical processes assigned to each of the processors.

In general, we see that the lock-free algorithm performs much better than CCT as the number of LPs per processor is small. As the number increases, the blocking effect in CCT is alleviated and the overhead of the lock-free algorithm due to extra complexity in bookkeeping begins to show. The overall performance caused by the overhead ranges from about 10% in light workload to less than 4% in moderate to relatively heavy workload when the number of LPs per processor is large. There are two important messages from the result. First, as the connectivity of the LP graph increases, the difference between the two algorithms increases. With small number of LPs per processor, the lock-free algorithm increases its performance advantage as the connectivity increases. This is because under higher connectivity, the slow process in the CCT algorithm is

able to block more of its neighbors and further slow down the system. On the other hand, with large number of LPs per processor, the opposite seems to be true where the lock-free algorithm gets slower for large number of connections in the LP graph. This is the result of increasing overhead of our algorithm. Our algorithm requires access to many variables (both shared and private) to help avoid unnecessary context switches. The overhead is proportional to the number of connections. Also, note that the crossing point shifts towards more LPs as the connectivity increases. Second, with increasing workload, the lock-free algorithm outperforms its counterpart in greater scale when the number of LPs is small. This is reflected by steeper curves as the event density is larger.

Figure 4 shows the ratios of context switches as we fixed the connection radius at 6. The curves coincide with the execution time ratios and show larger savings for the lock-free algorithm when the number of LPs is low. Also, it appears that the larger the event density, the smaller the number of context switches for the lock-free algorithm compared to CCT. As the savings in context switches for the lock-free algorithm decreases as we increase LP population, the overhead from the complexity of extra bookkeeping to avoid unnecessary context switches in the lock-free algorithm cannot be fully compensated any more. With more LPs ready for execution at any time on a processor, there will be a prolonged queueing time for all LPs on average. This makes it more likely that the remaining critical channels will be updated and therefore the unnecessary context switch will be avoided. That is, even though CCT does not try to avoid unnecessary context switches, multiprogramming certainly helps it avoid the issue with increasing number of processes. This further justifies the claim of CCT's authors to support simulation of low-granularity network models, which usually consist of large number of logical processes.

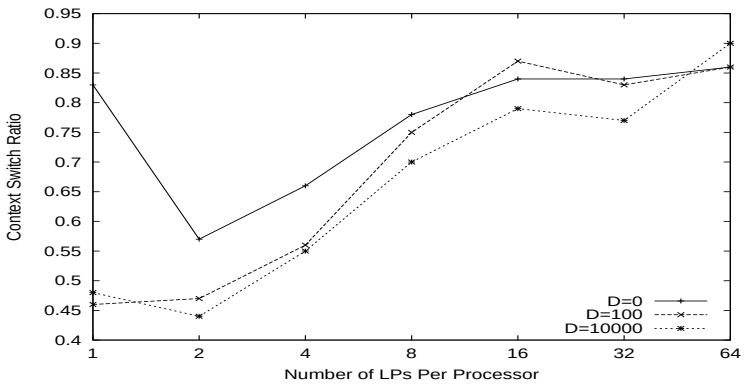

Figure 4: The ratio of the number of context switches of the lock-free algorithm to that of the CCT algorithm with increasing number of logical processes assigned to each of the processors. The connection radius is fixed at 6.

7 Conclusion

In this paper, we described a novel lock-free algorithm for scheduling logical processes in conservative parallel discrete-event simulation. We consider our algorithm as robust when compared with the algorithm using locks. In models with the number of LPs comparative to available processors, the algorithm shows much better performance and does not suffer from the defect of those algorithms using locks, where slow process

may block other processes and form a bottleneck that degrades the performance of the entire system. The benefit, though, comes with a cost. The added complexity of the lock-free algorithm introduced some overhead from manipulating variables used for bookkeeping. The overhead, though, has only modest effect on the overall performance of the algorithm especially when the workload becomes non-trivial.

References

[1] K. M. Chandy and J. Misra. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–52, 1979.

[2] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[3] P. Jayanti, A. Sethi, and E. L. Lloyd. Minimal shared information for concurrent reading and writing. *Distributed Algorithms. 5th International Workshop, WDAG '91. Proceedings*, pages 212–28, 1991.

[4] D. Jefferson and P. Reiher. Supercritical speedup (discrete event simulation). *Proceedings of the 24th Annual Simulation Symposium*, pages 159–68, 1991.

[5] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–25, 1985.

[6] J. Liu, D. Nicol, B. Premore, and A. Poplawski. Performance prediction of a parallel simulator. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 156–64, 1999.

[7] E. Naroska and U. Schwiegelshohn. A new scheduling method for parallel discrete-event simulation. *Proceedings of European Conference on Parallel Processing EURO-PAR '96*, pages 582–93 vol.2, 1996.

[8] D. Nicol, J. Liu, and J. Cowie. Safe timestamps and large-scale modeling. *Proceedings 14th Workshop on Parallel and Distributed Simulation (PADS 2000)*, pages 71–8, 2000.

[9] D. M. Nicol. Principles of conservative parallel simulation. *Proceedings of 1996 Winter Simulation Conference Proceedings*, pages 128–35, 1996.

[10] D. M. Nicol. Parallel discrete-event simualtion: so who cares? *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, 1997.

[11] Ha Yoon Song, R. A. Meyer, and R. Bagrodia. An empirical study of conservative scheduling. *Proceedings 14th Workshop on Parallel and Distributed Simulation (PADS 2000)*, pages 165–72, 2000.

[12] J.D Valois. Lock-free data structures. *Ph.D Thesis, Rensselaer Polytechnic Institute*, May 1995.

[13] Wagner D.B., Lazowska E.D., and Bershad B.N. Techniques for efficient shared-memory parallel simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 1989.

[14] Z. Xiao, B. Unger, R. Simmonds, and J. Cleary. Scheduling critical channels in conservative parallel discrete event simulation. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 20–8, 1999.