Addressing Blocking and Scalability in Critical Channel Traversing

Rob Simmonds, Cameron Kiddle and Brian Unger
{simmonds,kiddlec,unger}@cpsc.ucalgary.ca
Dept. Computer Science,
University of Calgary,
Alberta, Canada.

Abstract

This paper presents two new versions of the Critical Channel Traversing (CCT) algorithm. CCT is a conservative parallel discrete event simulation algorithm that has been shown to achieve very high performance when used in a wide area computer network simulator. The first of the new algorithms called simple sender side CCT *is similar to the original, but busy waiting is eliminated. Results presented show that simple sender side CCT avoids performance problems that can be caused by busy waiting.*

The second new algorithm called receive side CCT *employs a different strategy for updating channel clocks and determining which objects should be scheduled on critical channels. Performance results show that this version provides better scaling with respect to the connectivity of the model, at the expense of some added complexity.*

Keywords: *Parallel Discrete Event Simulation, Conservative Algorithms, Performance.*

1 Introduction

This paper presents two modified versions of the Critical Channel Traversing algorithm (CCT) [11]. The first is a simplification of the original that avoids busy waiting that could inhibit the performance of the original algorithm in some situations [7]. The second uses different data structures to store safe-time and scheduling information to enable improved scalability with respect to the connectivity of the model.

CCT is an extension of the Chandy-Misra-Bryant (CMB) [1, 2] parallel discrete event simulation (PDES) algorithm that incorporates scheduling decisions into the causality correctness calculations. Systems are modeled as a set of objects that only communicate by exchanging timestamped event messages. All messages sent from one object to another must be passed along a uni-directional communication *channel* that will usually have been allocated prior to the start of the simulation. Each channel has a *delay* associated with it representing the smallest difference in the clock of the sending object and the timestamp given to any event sent on the channel. Each channel also has a clock representing a lower bound on the timestamp of any event that could arrive on this channel in the future.

The main idea of CCT is to only schedule objects when it is likely they will have work to do. In a system using the CMB approach to guarantee the correct ordering of events, an object can only do work when its minimum channel clock increases. In CCT the input channel with the smallest clock value is marked as critical when an object completes an execution session, having executed all events with timestamps less than the clock of this channel. When the source object to the channel completes its execution session, it observes that the channel is marked critical and schedules the destination object for execution by inserting it into a scheduling queue. This avoids wasting processor time executing objects waiting for channel updates and also keeps the scheduling queues short making them less expensive to sort.

The original CCT algorithm along with the TasKit kernel in which it was implemented, were developed to provide a very high performance simulation engine for the Asynchronous Transfer Mode Traffic and Network Simulator (ATM-TN) [10]. This work built on previous work with similar aims [3]. While the original CCT performed well for what it was designed, it was clear that some simplification was possible and that greater scalability with respect to the connectivity of the model could be achieved. Since ATM network models are sparsely connected, scaling was not an important issue in the original design.

The *simple sender side (SSS) CCT* algorithm is very similar to the original CCT, but the use of busy waiting has been eliminated. The original CCT used busy flags to indicate when an object was processing events. Adjacent objects would wait for the candidate object's busy flags to be cleared before making scheduling decisions regarding the candidate. This resulted in the favorable semantic property

Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS'02)
1087-4097/02 $17.00 © 2002 **IEEE**

that any object with one or more input channels would always be scheduled by an object connected to one of these input channels. Unfortunately, the setting and clearing of busy flags added overhead and could also lead to excessive waiting. These properties resulted in poor performance for certain classes of models as described in [7].

The *receive side (RS) CCT* algorithm changes the way in which channel clocks are updated. This version adds some additional overhead for sparsely connected models, but greatly improves the way in which the performance of CCT scales with the connectivity of the model. This version aims to avoid accessing channels when they are not vital to the current causality or scheduling decision calculation.

The original and updated versions of the CCT algorithm have all been implemented in a new experimental simulation kernel called CCTKit. CCTKit shares the same general architecture as TasKit, but has been written from the ground up to achieve favorable cache behavior on modern shared-memory multiprocessor computers. Performance results are presented in this paper for simulations using a synthetic workload model. These results demonstrate the performance of the algorithms with models of different connectivities and different event densities. These results demonstrate the scaling properties and operating overheads of the new algorithms compared to the original.

The remainder of the paper is laid out as follows. Section 2 explains some of the important algorithmic and implementation concepts employed by the algorithms. Then, section 3 outlines the original and describes the two new versions of CCT. Section 4 describes experiments performed to compare the performance of the three algorithms and presents performance results. The paper is summarized in section 5.

2 CCT Components

This section explains the major ideas used in the original CCT algorithm. Similar ideas are used in the updated algorithms which are explained in more detail in Section 3.

2.1 Critical Channels

Each channel has a clock, which is a lower bound on the timestamp of any event that will be inserted into the channel in the future. A critical channel is a channel that has to have its clock advanced in order for the object at the receiving end of the channel to make progress. In CCT, an object which has one or more input channels sets the critical flag in one of these channels at the end of its execution session. The critical channel is a channel with the lowest clock. Only one channel is set critical even if many channels share this lowest clock value.

After setting a critical channel the object checks if any of its output channels have their critical flag set. When an output channel with its critical flag set is found, the channel's destination object is scheduled for execution. This is done by inserting the object into a processor scheduling queue ready to be executed when a processor becomes available. In the experiments presented in Section 4 each processor has its own scheduling queue. With this configuration in CCTKit, a FIFO single reader/single writer queue is allocated for each pair of processors that could communicate during a simulation. To schedule an object allocated to a different processor, a message is sent via the appropriate FIFO queue to request the processor controlling the object to be scheduled to place the object in its scheduling queue. Each processor scans its input queues between object execution sessions.

2.2 Channel Event Sampling

Only one event is removed from an input channel each time a channel is accessed. This event is then placed into the local event queue of the receiving object. The next event to be executed is always removed from the local event queue which is a priority queue sorted into increasing event timestamp order. An event that arrived from a channel contains a reference to the channel it was received on. After each event is executed, if that event arrived via a channel, an attempt is made to remove another event from that same channel. If another event is recovered, this event is placed into the local event queue as before. If no event is recovered from the channel, the channel clock is noted and incorporated into the safe-time calculation (see Section 2.3).

Always processing the lowest timestamped event from the local priority queue ensures local event ordering. Placing at most one event from any input channel into the local event queue at any time keeps the priority queue short. This is important since the priority queue will have costs proportional to its length for operations performed on it. All operations on the channels, which implement single reader/single writer FIFO queues, are O(1).

2.3 Safe Time Calculation

The object's safe-time is the time value that is known to be the lower bound on the timestamp of any event that could arrive from another object in the future. This is calculated as the minimum input channel clock value. Rather than calculating the safe-time once at the start of an execution session, an initial safe-time estimate determined and refined during the session.

At the start of an execution session the safe-time estimate is set to ∞. The input channels are then scanned and if they are empty and don't currently have a representative event in

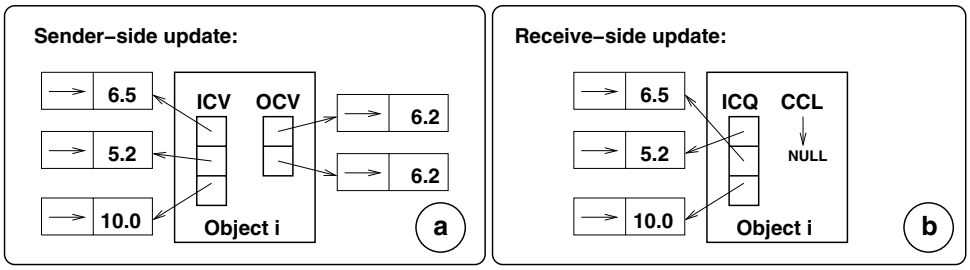

Figure 1. Data structures used by sender-side (left) and receive-side (right) CCT.

the local event queue, the channel clock is included in the safe-time calculation. If no empty channel is encountered it is possible that an object could continue to execute until the simulation end-time is reached. This is very unlikely to occur in practice since it would rely on all of the objects with channels connected to this object advancing fast enough for there always to be an event to remove from a channel when the previous event from that channel has been executed.

Note that if an object does not have input channels the safe-time would always be infinite using the rules above. Allowing these source objects to execute until the end-time could lead to buffer exhaustion if other objects are not executing fast enough to consume events being dispatched. There are several solutions to this problem including simply eliminating source objects by adding extra channels or by using a flow-control mechanism. This issue is addressed in [9].

3 CCT Algorithms

This section explains three versions of the CCT algorithm all of which are implemented in CCTKit. These are *blocking sender-side* (BSS) CCT which is the original CCT algorithm, *simple sender-side* (SSS) CCT and *receive-side* (RS) CCT. A proof of correctness for BSS was presented in [11] and the proof correctness of SSS and RS follow from that.

The algorithms are explained assuming that all objects have at least one input channel. If an object has no input channels, a different mechanism is required to terminate an execution session. The algorithms can be used as they are presented with models containing objects with no input channels by adding extra channels feeding back to the source objects. Other mechanisms for dealing with source objects are discussed in [9].

3.1 Sender-Side CCT

Blocking Sender Side CCT

This section gives a brief recap on the original CCT algorithm which is described as *blocking sender side* (BSS) CCT in this paper. Each object involved in the BSS calculations has an input channel vector (ICV) containing all of its input channels, and an output channel vector (OCV) holding all its output channels (see Figure 1a). The actions performed during each object execution sessions are outlined in Figure 2.

1. Scan input channels setting busy flag on each channel and calculating the initial safe-time estimate.
2. Execute events up to safe-time getting events from channels and revising the safe-time estimate as necessary.
3. Set critical channel.
4. Unset busy flag on each input channel.
5. Scan each output channel updating the channel clock. For each channel check the busy flag and wait while this is set. Then schedule the destination object if the critical flag is set.

Figure 2. Pseudo code for a blocking sender side (BSS) CCT object's execution session.

An execution session begins when the object is removed from the scheduling queue by the simulation kernel. First, all of the input channels are scanned and an initial estimate of the safe-time (see Section 2.3) is calculated (line 1). During this scan the busy flag is set on each of the input channels. Next a loop is entered where events are executed and new events are recovered from the input channels un-

til all events have been executed with timestamps less than the minimum clock of an empty channel (line 2). Then the minimally clocked channel found in the previous step has its critical flag set (line 3). Now each of the input channel busy flags are cleared (line 4). Next each output channel is accessed and it's clock updated to be the minimum of object's clock plus the channel delay and the timestamp of the last event sent on this channel. After updating the clock, the busy flag is checked and if it is set, the object waits until it is cleared. At this point the destination object is scheduled if the critical channel is set (line 5).

In order to work correctly it is vital that each busy flag is set before the channel clock is incorporated into the initial safe-time estimate calculation. If not, it is possible that the object at the sending side of the channel could fail to observe the critical flag and therefore fail to schedule this object. It is also vital that the channel clock is updated before the critical channel flag is checked. For computers that do not support a sequential consistent memory model [4], memory ordering instructions are inserted between these pairs of actions.

Simple Sender Side CCT

The chief difference between *simple sender side* (SSS) CCT and the BSS CCT described above is that busy waiting is not performed in SSS CCT. Eliminating busy waiting means that it is now possible for an adjacent object to complete its execution session without observing a critical channel leading to this object. Therefore an additional check is required to determine if this has occurred so that the object can schedule itself. The data structures used for SSS are the same as those used for BSS (see Figure 1a). The actions performed during an object's execution session are outlined in Figure 3.

An SSS object execution session starts in a similar way as a BSS execution session. The input channels are scanned to find the initial safe-time estimate (line 1). In this case no busy flags are set. The execution of events is identical to BSS (line 2). Next each output channel is visited, its clock updated and the destination object scheduled if the critical channel is set (line 3). Note that unlike the BSS case, the critical channel has not yet been set. This is done in the final step (line 4). In this case the critical channel is only set if the channel clock on the candidate channel has not been updated beyond what it was when the loop on line 2 was exited. If it has advanced, this is an indication that this channel's source object has just executed and would have scheduled this destination object at the end of its execution session if busy waiting had been used. Therefore, rather than setting the critical channel this object schedules itself. If the channel clock has not advanced, the critical channel flag is set.

1. Scan input channels to discover initial safe-time estimate.
2. Execute events up to safe-time, getting events from channels and revising the safe-time estimate as necessary.
3. Scan each output channel updating the channel clock and scheduling the destination object if the critical channel flag is set.
4. If time has not advanced on candidate channel set critical flag on this channel. If time has advanced, schedule self.

Figure 3. Pseudo code for a simple sender side (SSS) CCT object's execution session.

Similar memory ordering considerations have to be made for SSS as were made for BSS. An important point in this case is in the setting of the critical channel. It is vital that the source object does not miss the setting of the critical channel and the destination object miss the update of the channel clock at line 4. To prevent this from occurring, the critical channel flag is always set, then a memory ordering instruction performed if running on a computer that does not support sequentially consistent memory access. After this the channel clock is checked. If the channel clock has advanced the critical channel flag is cleared before the object schedules itself. These actions insure that at least one of the source and destination objects schedules the destination object when required.

3.2 Receive-Side CCT

Receive-side (RS) CCT uses different data structures than the BSS and SSS versions. For RS CCT, the input channel vector is replaced with an *input channel queue* (ICQ) and the output channel vector is replaced with a *critical channel list* (CCL) (see Figure 1b). The aim of the RS CCT algorithm is to avoid checking channels that are not currently relevant to safe-time or scheduling calculations. The actions performed during an object's execution session are outlined in Figure 4.

The initial safe-time estimate is not calculated by scanning the input channels. Instead, it is set to the clock of the channel at the top of the ICQ if the queue is not empty and set to ∞ otherwise (line 1).

Next all events in the local event queue with timestamps less than the current safe-time estimate are executed (line 2). Each time an event is executed that arrived on an input

1. Set the initial safe-time estimate as either the clock of the channel at the top of the ICQ or ∞ if the ICQ is empty.
2. Execute all events from local event queue with timestamps less than the safe-time. For each event received from a channel, attempt to remove another event from the same channel and place it in the local queue. If there is no event in the channel, place the channel in the ICQ and revise safe-time to the clock of this channel if it is less than the current safe-time estimate.
3. Attempt to get event from queue at top of ICQ. If successful, update the channel clock using the event's timestamp and remove the channel from the ICQ. Otherwise attempt a receive-side clock update. If the clock cannot be advanced move to stage 4. Otherwise resort the ICQ. If the new safe-time (clock of channel at top of ICQ) is at least as great as the timestamp of the next event in the local event queue, repeat stage 2. Otherwise repeat stage 3.
4. Schedule all objects held in the CCL.
5. Instigate placing self into the CCL of the source object to the channel at the top of this object's ICQ

Figure 4. Pseudo code for a receive side (RS) CCT object's execution session.

channel, an attempt is made to recover another event from the same input channel. If an event is recovered, it is inserted into the local event queue. Otherwise, this channel is inserted into the ICQ and the safe-time estimate set to the clock of this channel if its clock is less than the current safe-time estimate. This phase of execution continues, executing events and attempting to remove new events from input channels until the safe-time is reached.

The next phase attempts to increase the safe-time by updating the clock of the top channel in the ICQ. The channel clock is advanced either by removing an event and setting the clock to the event's timestamp, or by performing a receive side update. A receive side update sets the channel clock to the sum of the clock of the source object plus the channel's delay. This phase completes either by looping back to the event execution phase (line 2) or by advancing to the scheduling phase (line 4).

First an attempt is made to remove an event from the channel at the top of the ICQ. If this is successful the event is placed in the local event queue, the channel removed from the ICQ and its clock updated. If there is no event in the channel a receive side update is performed and if this results in the channel clock increasing, the ICQ is resorted. If either an event was recovered or the receive side update resulted in the channel clock advancing, a new safe time is determined as the clock of the channel now at the top of the ICQ, or ∞ if the ICQ is empty.

At this point the execution returns to the event execution phase (line 2) if the safe-time is greater than or equal to the timestamp of the event at the top of the local event queue, or returns to the start of the current phase (line 3) otherwise. If the attempt to increase the channel clock failed, the algorithm moves to the scheduling phase (line 4). During the scheduling phase all objects in the CCL are scheduled for execution.

Finally the object moves to the critical schedule request phase (line 5). In this phase a request is made to insert the calling object into the CCL of the source object of its critical channel. The critical channel is the channel at the top of the ICQ. If the source object is allocated to the same processor as the calling object, the local processor inserts the object immediately. If the source object is allocated to a different thread, a message is sent to the thread controlling the source object. When this message is received, a check is performed to determine if the clock of the critical channel has advanced since this channel became critical. If so the destination object is scheduled immediately. Otherwise, the destination object is inserted into the source objects CCL ready to be scheduled during the source object's next execution session.

One problem with RS CCT is that the channel clocks are only updated using event timestamp and sender clock plus channel delay information. As described here, RS CCT cannot take advantage of sender-side channel clock updates greater than the maximum of these two values that may be required by a system performing lookahead optimization. Ways in which lookahead optimizations can be supported with the minimum of additional overhead are being investigated.

4 Performance

This section describes the experiments performed to evaluate the performance of the various CCT algorithms.

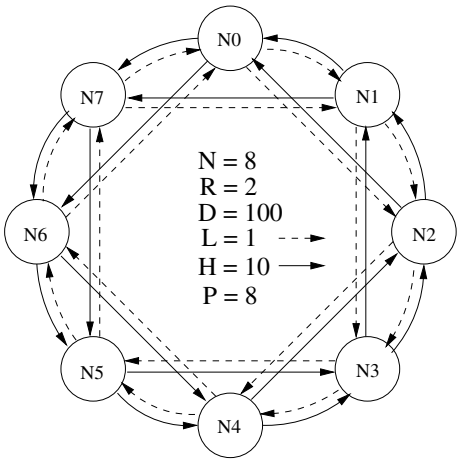

Figure 5. Ring model with $N = 8$ nodes (objects), a connectivity radius of $R = 2$ and an initial event population on each node of $D = 100$. Half of the channels have a delay of $L = 1$, and half of the channels have a delay of $H = 10$. The model is configured to run on $P = 8$ processors (i.e., 1 node per processor).

It explains the workload model used, the experimental methodology and presents results of the experiments.

4.1 Synthetic Ring Workload

A ring model similar to the model presented in [7] was used for experiments for which results are presented in this section. The model is parameterized with: *N* - the number of nodes (objects), *R* - the connectivity radius, *D* - the event population, *L* - the low channel delay, *H* - the high channel delay and *P* - the number of processors (see figure 5).

Each node is connected to R nodes ahead in the ring with low channel delay L and R nodes behind in the ring with high channel delay H. A model has $N \times 2R$ channels for values of R from 1 to $\frac{N}{2} - 1$ and $N \times 2R - N$ channels for the fully connected case where $R = \frac{N}{2}$. The difference in the number of channels is explained by the same node being both R steps ahead and R steps behind in the latter case. Initially, each node is populated with D events having exponentially distributed inter-arrival times with mean $\frac{L}{D}$.

Upon receiving an event each node sends a new event on a randomly selected output channel with a timestamp equal to the timestamp of the event received plus the channel delay. Once the initial D events per node have been generated the number of events circulating in the system remains constant at $N \times D$. If D is zero then no events are generated during the simulation run. The N nodes are statically allocated to the P processors such that the ith node is allocated to processor $\frac{i \times P}{N}$.

4.2 Results

This section presents results of experiments performed to demonstrate the performance of the various CCT algorithms. An 8-processor Compaq Prolient server with 700MHz Intel PIII Xeon processors was used for all tests. The Prolient was running RedHat Linux with the v2.4.9 kernel. The GNU *g++* V2.95.3 was used with the *-O2* optimization flag. The Pthreads library [6] was used to provide concurrent execution.

Each test was run for 60 seconds of wall-clock time. At the end of each test the minimum simulation time reached by all objects was recorded. Each test was run 10 times and the mean of the recorded values and a 95% confidence interval was calculated. Additional tests were performed for the few cases where the 95% confidence interval was greater than 10% around the sample mean. The relative performance of the algorithms was set to 1 if the confidence intervals of the corresponding results overlapped.

The two graphs in Figure 6 show the relative performance of SSS CCT to BSS CCT for cases where the performance of BSS is inhibited by busy waiting. In both cases the tests were run on 8 processors with the number of nodes (N) varied from 16 to 256. For these tests both the low (L) and high (H) channel delays were set to 1. The graph on the left shows the relative performance when an event population (D) of 100 and three different values of the connection radius (R) are used. The graph on the right shows the relative performance when the connection radius (R) was set to 4, for values of the event population (D) of 10, 100 and 1000.

With BSS CCT, the larger the number of nodes, the lower the chance of a processor being blocked by another processor working on an adjacent object. Therefore, for larger values of N, the closer the performance of the two algorithms. For both sets of experiments, the performance of both algorithms is identical when 128 or more nodes are used. With BSS CCT, the greater the connection radius (R), the greater the chance of blocking. Therefore, there is a greater difference between the performance of SSS CCT, which does not use busy waiting and BSS CCT for smaller R values.

With BSS CCT, increasing the workload could result in a processor being blocked for longer periods of time. The graph on the right shows that the advantage gained by using SSS CCT is greatest when the event population is high.

The graphs in Figure 7 compare the performance of RS CCT to the performance of SSS CCT. These tests were performed on 8 processors with connectivity radius (R) values between 1 and 32. For these tests 1024 nodes (N) were used and the low channel delay (L) was set to 1. The graph on the left shows the relative performance with an event popu-

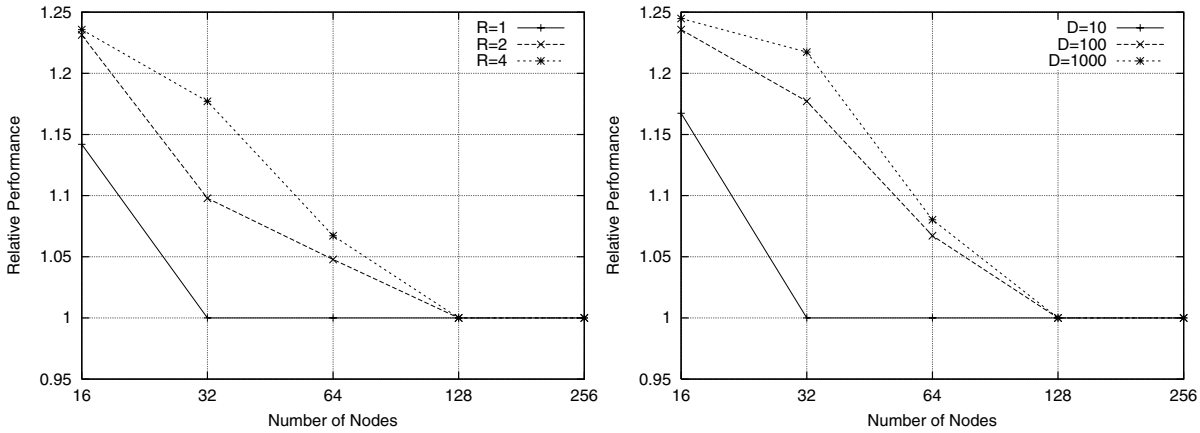

Figure 6. Results showing the relative performance of SSS CCT to BSS CCT versus the number of nodes (N) for L=H=1 and P=8. The left graph shows results for different values of R with D=100. The right graph shows results for different values of D with R=4.

lation (D) of 0 and three different values of the high channel delay (H). The graph on the right shows the relative performance when the high channel delay (H) was set to 10, for values of the event population (D) of 0, 10 and 100.

Increasing the high channel delay reduces the work done by RS CCT since the channels that are given high clock values are accessed less often. For SSS CCT all the channels have to be accessed in each execution session whatever their delay and clock values. From the left graph it can be seen that for greater values of H the advantage gained using RS CCT increases.

From the right graph it is clear that at higher connectivities the advantage gained by using RS CCT is reduced at higher event densities. This is due to a greater proportion of work being done executing events than handling channel state for larger values of D. This graph also shows that when events are being received, there is some overhead using RS CCT at low connectivities. This is most likely caused by the cost of adding and removing channels from the ICQ. This result suggests that some more experimentation may be required to determine the best strategy for handling the channels in the case when events are being handled.

Initial testing of these algorithms within a computer network simulator suggests that RS CCT will perform at least as well as SSS CCT for a large class of network models. This is likely explained by the large differences in the channel delays in these models. For example, consider an object representing a backbone router. This will have some links connecting to other backbone routers and some links connecting to border routers. The channels used to represent links between backbone routers have large delays, since the distance between these routers is large. The channels used to represent links to border routers have much smaller delay values. Therefore, the set of channels that have small delays and that will need to be accessed often using RS CCT will be small at many of the simulation objects. As shown in Figure 7 (left), this improves the performance of RS CCT compared to SSS CCT.

5 Summary

This paper has presented two new versions of the Critical Channel Traversing (CCT) algorithm [11]. Simple sender side CCT is similar to the original CCT, but busy waiting has been eliminated. Receive side CCT uses new methods for finding the safe time and for scheduling objects connected to critical channels.

Results presented show that simple sender side CCT performs at least as well as the original algorithm in the majority of cases. They also show that it avoids performance problems that the original algorithm encountered with certain classes of models. Receive side CCT is shown to scale far better with respect to the connectivity of the system being modeled, than either of the other CCT versions discussed in this paper. It does this at the expense of some additional overhead for low connectivity models. It is hoped that further research refining both the algorithm and its implementation will reduce these overheads.

The refinements presented in this paper improve the CCT algorithm, an algorithm previously shown to be a good choice for use in a wide area computer network simulator. While these refinements make CCT more flexible, it should still be noted that good performance from any channel based conservative simulator is dependent on models with good lookahead and large event densities. Models that do not have these properties could be better suited to simulators implementing synchronous conservative [8] or opti-

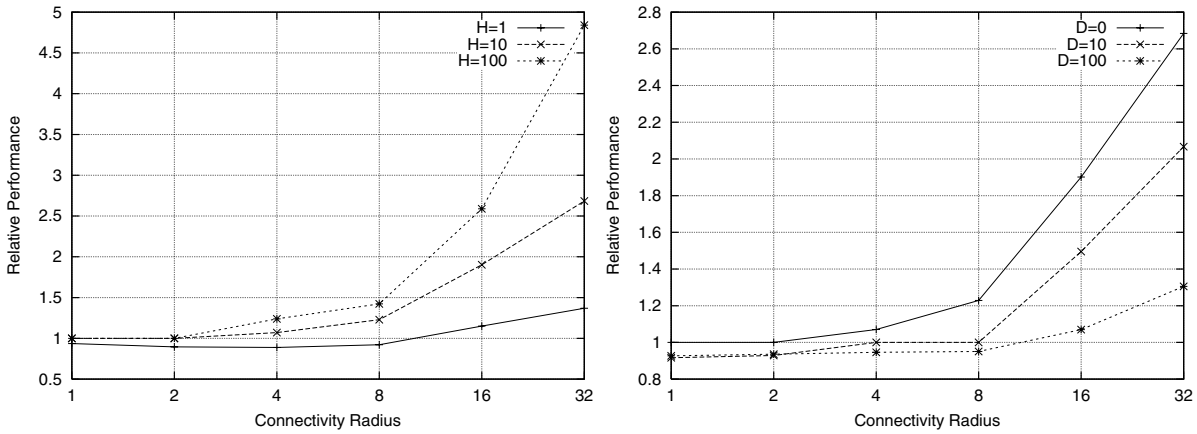

Figure 7. Results showing the relative performance of RS CCT to SSS CCT versus the connectivity radius (R) for N=1024, L=1 and P=8. The left graph shows results for different values of H with D=0. The right graph shows results for different values of D with H=10.

mistic [5] algorithms.

6 Acknowledgments

The TeleSim group is funded by grants from the Canadian Natural Sciences and Engineering Research Council (NSERC) and the Alberta Science and Research Authority (ASRA). TeleSim also receives funding from industrial sponsors including Nortel Networks, Telus, Siemens and Compaq. Experiments for which results are presented in this paper were performed on computers purchased as part of the MACI high performance computing project.

References

[1] R. Bryant. Simulation of Packet Communication Architecture Computer Systems. Technical Report MIT/LCS/TR-188, MIT, November 1977.

[2] K. M. Chandy and J. Misra. Distributed simulation : A case study in design and verification of distributed simulation. *IEEE Transactions on Software Engineering*, 5(5):440–452, September 1979.

[3] John G. Cleary and Jya-Jang Tsai. Performance of a conservative simulator of ATM networks. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 142–145. The Society for Computer Simulation, 1997.

[4] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture : A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[5] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, pages 404–425, July 1985.

[6] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads*. Sun Microsystems, 1997.

[7] Jason Liu, David M. Nicol, and King Tan. Lock-free scheduling of logical processes in parallel simulation. In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, pages 22–31, May 2001.

[8] David M. Nicol. Principles of conservative parallel simulation. In *Proceedings of the 1996 Winter Simulation Conference*, pages 128–135. The Society for Computer Simulation, 1996.

[9] Rob Simmonds, Cameron Kiddle, Kitty Wong, and Brian Unger. Controlling buffer usage in critical channel traversing. In *2002 Advanced Simulation Technologies Conference*. ACM, April 2002.

[10] Brian W. Unger, Fabian Gomes, Xiao Zhonge, Pawel Gburzynski añd Theodore Ono-Tesfaye, Srinivasan Ramaswamy, Carey Williamson, and Alan Covington. A High Fidelity ATM Traffic and Network Simulator. In *Proceedings of the 1995 Winter Simulation Conference*, pages 996–1003. The Society for Computer Simulation, 1995.

[11] Z Xiao, B Unger, R Simmonds, and J Cleary. Scheduling Critical Channels in Conservative Parallel Discrete Event Simulation. In *Proceeding of the 13th Workshop on Parallel and Distributed Simulation*, May 1999.