Abstract

A fundamental problem in parallel and distributed discrete event simulation is to guarantee causal event ordering for each and every entity in the simulation system. A conservative synchronization algorithm CCT (Critical Channel Traversing) has demonstrated phenomenal performance in the ATM network simulation unseen before. Some of the major factors that contribute to the performance are the ability of CCT to schedule simulation at the largest possible computation grains and the use of a set of lock-free concurrent control mechanisms.

This paper presents the features and properties of CCT from the performance perspective. It also describes the concurrent control mechanisms that are designed to work with CCT to support fast simulation of large communication networks.

1 Introduction

High bandwidth communication networks such as ATM networks contain very large volume of messages. Simulation of these types of networks typically involves a large number of events (e.g., billions to trillion of ATM cells). Compared to conventional sequential simulation, parallel simulation can provide significant speedup. However, the fine granularity and dynamic nature of events involved in the communication network simulation present great challenge.

We have been working on high performance parallel simulation of communication networks based on Parallel Discrete Event Simulation (PDES) technique, in particular the modeling and simulation of ATM networks [7]. This paper is focused on the performance issues involved in the underlying support system for PDES — the simulation kernel.

In the rest of this section, the background of PDES and the two major approaches to synchronization are briefly described. Furthermore, we recall how our experience in pursuing fast ATM network simulation eventually led to the design of a new algorithm and its implementation in a simulation kernel.

Discrete event simulation assumes that the system being simulated only changes state at discrete points of time upon the occurrence of an event. The system being modeled is viewed as being composed of a number of entities that interact at various points in time. The simulation system models physical entities as logical processes and interactions between entities as time-stamped event messages sent to each other [4]. Most of the modern discrete event simulation methods are based on this logical process view.

The basic problem in PDES is that the causal relations between events must be maintained in order for the computation to be correct. In sequential simulation, traditionally, the smallest time-stamped event from the event list is the one to be processed next. This global event ordering is overly sufficient. It can be proven that it is sufficient for keeping causality relationship between events if each Logical Process (LP) processes events in non-decreasing timestamp order. This is referred to as Local Causality Constraint (LCC) [4].

Parallel synchronization algorithms for ensuring LCC generally fall into two categories: conservative [3, 11 and optimistic [5]. Conservative approaches strictly avoid the possibility of any causality errors ever occurring. Optimistic approaches allow causal errors in exchange for a better chance to exploit parallelism. Errors are detected and a rollback protocol is used to recover from erroneous computations. Most of the terms and concepts introduced above refer to [4], which is a good source on PDES.

We have experimented with both approaches in the ATM network simulation [7]. As a starting point, we defined a C++ library, SimKit [6], which provides an interface to functions in the simulation kernel. These kernel functions implement synchronization algorithms in such a way that users of SimKit need not be concerned with the details of how events
are scheduled to maintain LCC. For most part, the same simulation model can be run on different kernels without change. Figure 1 shows the architecture of simulation softwares.

![Simulation software architecture](image)

Figure 1: Simulation software architecture

Three kernels have been built and tested with a set of ATM network scenarios of varying traffic loads. They are: a central event list based sequential kernel CelKit, a Time Warp based optimistic parallel kernel WarpKit [11], and a conservative parallel kernel WaKit [2]. The performance results of those tests are reported in [8]. WarpKit is able to provide robust performance with speedup of 4 to 5 relative to the sequential kernel on 16 processors. However, the high system overheads in saving state and rollback limit its performance. WaKit is designed to have very low system cost and is specifically tuned to the characteristics of cell-level ATM simulation. It performs extremely well on one network scenario (on single processor, it is faster than CelKit). But it is susceptible to small changes in the network model and is outperformed by the optimistic kernel, WarpKit, on most of the test scenarios.

In conservative algorithms, it has to be determined for a LP if the next event can be processed safely without violation of LCC. LPs that contain no safe events must be blocked until they are available. Finding which LPs have safe events and are ready for execution is a strategy that the simulation kernel has to decide on. It is traditionally not part of the parallel synchronization algorithm. WaKit uses a single static list of all the LPs assigned onto a particular processor. This list is repeatedly scanned and each LP in turn scheduled. This gives very low overhead so long as each LP has events to process on each cycle. Computation locality can also be improved with each LP processing as many events as allowed before moving onto next LP. As a result, WaKit can run faster on single processor than CelKit on some test scenarios. But, it performs very poorly with other scenarios where few events are processed on each cycle.

Our experiments with WaKit led to the belief that conservative approach can perform much better in communication network simulations because of the scarce connectivity (a switch only connects to its neighboring switches) and high message density. To utilize this performance potential, we need a new conservative algorithm, one that not just satisfies LCC, but also schedules LPs dynamically in a way that is able to maximize the number of safe events.

The new algorithm is called Critical Channel Traversing (CCT) [10]. A kernel that implements the algorithm has demonstrated excellent performance. For the ATM network scenarios which we have tested with three previous kernels, the new kernel, called TasKit, has achieved two to four time speedup on a single processor with respect to CelKit. On a 16 processor shared-memory SGI Power-Challenge, a speedup of 26 times has been observed on a larger ATM network model [9].

This paper discusses the features and properties of CCT that enable high performance and the concurrent control methods that are designed to take advantages of the CCT algorithm to support fast simulation of large communication networks.

## 2 Multi-level Scheduling

The complete CCT algorithm and the proof of its correctness are covered in a previous paper [10]. In this paper we present CCT from the performance perspective.

CCT extends the logical process model of simulation through the addition of uni-directional channels. A channel is used to connect a sender LP to a receiver LP where messages can only be sent from the sender to the receiver. Each message carries a timestamp indicating the receive time of the message. We assume that messages sent by an LP to a given channel must follow non-decreasing timestamp order, which will be referred to as sending order.

CCT tries to schedule larger grains of computation by incorporating logical process scheduling into an event scheduling algorithm and makes its scheduling decision solely based on information local to LPs. In this section we give brief description of the scheduling strategy at each level starting from event scheduling and explain why CCT can be very efficient.

### 2.1 Scheduling Events

Messages are scheduled for processing when their destination LP executes, or is executed by a processor. While messages and events are used interchangeably in PDES literature, we refer to an event as a scheduled message, that is, a message has been put into an
event list local to the receiver LP. In this section, we look at how safe events are determined and scheduled.

An LP may have a number of input channels on which it can receive messages from other LPs and a number of output channels for which it can send messages to others. A channel has a clock CT and a delay δ. The value of clock is the lower bound on timestamps of any messages arriving at the channel in the future. The delay of a channel is the minimum time increment from the receive time of the event being processed by the channel's sender LP to the timestamp of messages, if any, sent to the channel. It is also called minimum lookahead on the channel and its value is application dependent. The transmission and propagation delay on a link between two switches is a good example.

We use RT to denote the lower bound on timestamps of any events an LP will receive in the future. BT is the simulation time up to which an LP has progressed, called local receive time (LRT).

At the start of a LP's execution session, the minimum of all its input channels' clock values, min(CT), is called safe-time ST. It can be seen, from the sending order assumption, no messages arriving at any input channels in the future will have timestamp less than ST. It follows that all the messages with timestamps equal to or less than ST currently held in input channels are safe events.

An LP's RT in the end of its execution session should be set to its safe-time, RT = ST. As ST also represents the lowest possible time of any events the LP will receive in the future. Let ST₁, ST₂, ..., STₖ be the sequence of safe-times in consecutive execution sessions 1, 2, ..., k respectively, then the local receive times of the LP will be RT₁, RT₂, ..., RTₖ and the advance of RT in session k will be STₖ - STₖ₋₁. Another important step in the end of a LP's execution session is to set the clock of each of its output channels to the highest lower bound on timestamps of future messages for that channel. That is, for output channel i = 1, 2, ..., n, n is the number of output channels, its clock value in the end of k session, CTᵢₖ is set to:

\[ CTᵢₖ = \max(tᵢₖ, CTᵢₖ₋₁, RTᵢ + δᵢ) \]

where CTᵢₖ₋₁ is the clock value in the last session k - 1, δᵢ is the delay on output channel i, tᵢₖ is the timestamp of the last message sent to channel i (if no messages have been sent to channel i in session k, tᵢₖ is not considered). Each item in (1) gives a lower bound on timestamps of future messages, so the one with maximum value is taken to be the channel's new clock value.

In the k-th session of an LP, all the events with timestamp \( t \geq RTₖ₋₁ (RTₖ₋₁ = STₖ₋₁) \) and \( t \leq STₖ \) are safe and must be processed in non-decreasing time order.

Figure 2 shows, before and after the execution session k of an LP, the input and output channels, messages and the local event list along with various clock values. CCT uses a sampling technique to order messages from different input channels and estimates safe-time at the same time, which requires to read the timestamp of a message only once. Instead of scheduling into the event list every safe message from all input channels, only one message from each input channel is placed in the event list. Each time an event has been processed, another message from the same channel as the one, from which the processed event was taken, will be scheduled into the event list. If the channel becomes empty, its clock value is compared with the current estimation of safe-time (initially STₖ is set to ∞) and the safe-time takes the lower value. STₖ will eventually come down to the lowest clock value of all input channels. The channel that holds STₖ will have no messages left. We call it critical channel because its receiver LP will not be ready for execution again until its clock advances. The concept of critical channels plays an essential role in the LP scheduling part of CCT algorithm.

In figure 2 left, input channel 2 was critical after last session k - 1, its first message (if any) in the current session k must be scheduled before any events can be processed out of the local event list. After session k, as shown in the right part of the figure, input channel 3 becomes critical and all messages con-
tained in it have been processed. The two events (with timestamps greater than $ST_k$) left in the event list are from channel 1 and 2 respectively.

Now we analyze the performance implications of the event scheduling algorithm described above. Each LP has its own local event list and the maximum length of the list is about the number of its input channels plus one extra possible event scheduled by the LP for itself. Simple linked list can be used for the event list and potentially very efficient because of its short length. This is one of the reasons why TaskKit usually outperforms CelKit even on single processor. Like other conventional sequential kernels, CelKit uses a centralized event list. The size of the list could become very large in large models with high message density. Test results on a large ATM model show that, with CelKit, the average real CPU time increases significantly with the traffic load because the length of queue increases and the queue operation becomes more costly. With TaskKit, the size of event list does not change with the size and load of the model.

Another important fact which contributes to the performance is that CCT tries to schedule LPs with the largest possible number of events. As a results, a processor executing a LP will work on the same set of objects before exhausting safe events. This can be translated into good locality of memory access and less per event share of system overhead. We will come back to this topic later.

### 2.2 Scheduling Logical Processes

A logical process repeatedly cycles through three states, ready, executing and waiting. An LP is ready when it has been scheduled but not yet chosen by a processor for execution. It is in executing if a processor is executing on it. After each execution session, an LP will be waiting to be scheduled again. We have covered LP execution, now we look at the issue of how LPs are scheduled in CCT.

One of the input channels of an LP will become critical in the end of an execution session. After updating the clock of an output channel by \( (l) \), the sender LP will schedule the receiver LP of that channel if and only if the channel is critical and its clock has advanced. LPs without input channels are called source LPs. A source LP can, theoretically, finish execution in a single session as its safe-time is \( \infty \). However, this is undesirable — large number of buffers will be occupied by messages with timestamp in the far future, which may exceed the space limit. A proper safe time can be set in each execution and the source LP schedules itself for the next session.

This seemingly very simple LP scheduling strategy has a potential impact on the performance of conservative method. Among other things, the algorithm is to schedule LPs before scheduling events; Event scheduling becomes local to LPs. Scheduled LPs either have a number of events to process without fear of causal errors, or at least will be able to advance its \( RT \). They are perfect parallel execution units, their execution do not have to be strictly ordered although it is often beneficial to chose LPs with lower \( RT \) to execute next.

We have described features of most part of the basic CCT algorithm. The properties of CCT discussed in next section further reveal its effect on performance.

### 2.3 Properties of CCT

We have learnt that an LP will advance its simulation time by \( \Delta ST_k = ST_k - ST_{k-1} \) in session \( k \). A larger time advance will result in more events processed in an execution and better performance. Now we show that CCT is able to extract, for each LP in every execution, the largest safe time that is available.

An LP, say \( LP_m \) as in Figure 3, will be scheduled for execution in session \( k \) only when the sender LP at its critical channel, let be \( LP_{m-1} \), executes. If one keeps tracking back all the critical channels, one will be able to find the very first LP, say \( LP_1 \), which has eventually caused \( LP_m \) to be scheduled. Given the finite number of LPs in a simulation, there can be only three possible cases as illustrated in a), b) and c) of figure 3. In a), \( \Delta ST_k \) at \( LP_m \) depends mainly on the advance of safe-time, \( \Delta ST \), at \( LP_1 \). Since \( LP_1 \) is a source LP, so its \( \Delta ST \) can be set sufficiently large, so becomes \( ST_k \). For b), \( LP_m \) is in a loop of LPs and \( LP_1 \) is \( LP_m \) itself, its \( \Delta ST_k \) is the sum of lookaheads on all channels in the loop, called Loop Time (LT).

Let \( CT_i \) be the clock value of \( LP_i \)'s output channel, then lookahead on that channel will be \( CT_i - RT \). As for c), \( \Delta ST_k \) is determined by \( \Delta ST \) at \( LP_1 \), which in turn is limited by the LT of the loop of LPs containing \( LP_1 \).

![Figure 3: Back-tracking critical channels](image)

It can be seen that the safe-time of any LP involved in an LP loop is limited by the LT of the loop. The value of LT changes from one loop to another and
even for different sessions of the same loop. For a given LP, the back trace of critical channels changes in different sessions as critical channels change. CCT is able to get the maximum safe time, which is restricted by \( LT \), for every LP in every session.

From formula (1), it can be derived that the minimum \( LT \) for any LP loop, called Minimum Loop Time, is the sum of delays on all the channels in the loop. \( MLT \) is an attribute of the application. Non-zero \( MLT \) for all the LP loops in a simulation is the essential condition for CCT to be free of deadlock.

2.4 Scheduling Tasks

The basic CCT algorithm can be further extended to schedule units of computation even larger than LPs. A set of LPs that have a high dependency on one another can be grouped into a task. A tasks is scheduled as a single unit in a way much like an LP is scheduled.

Currently, TasKit kernel recognizes two types of task, pipe-type task and cluster task. Pipe-task is used for LPs that are connected in a form of pipeline. Cluster task is used to group LPs that are closely related, for example, LPs in loops of zero or very low \( MLT \). This allows CCT to be usable to applications containing zero or low \( MLT \). Figure 4 shows two examples of tasks.

![Figure 4: Examples of pipe-task and cluster task](image)

In a pipe-task, messages output from one LP are input to next LP along the pipeline and the executions of LPs are in sequence. Memory locality can be greatly improved which could significantly speed up event processing due to caching benefit.

Rules of CCT apply to task scheduling and execution except that channels inside a task are not used for LP scheduling. In a cluster task, no LP scheduling is involved, a single event list is used for scheduling events for all LPs within the task. It is different in a pipe-task where execution starts from the first LP at the input side, then moves on along the pipeline.

2.5 Task Queues and Load Balancing

Most PDES kernels require the partitioning of LPs and mapping partitions to processors. Partitioning is difficult and more than often it results in unbalanced loads. Dynamic load balancing is much more difficult in fine granularity event simulations.

TasKit uses task queues to order ready tasks. If a centralized task queue is used, which is the case for our experiments with ATM network simulation. There is no mapping between tasks and processors. Each processor takes the next task for execution from the task queue. As long as there are enough ready tasks, load among processors are automatically balanced. For some SMP platforms, multiple task queues may yield better performance. Tasks are partitioned and assigned to different queues. Each processor owns a task queue. Work on load balancing strategies for this scheme is in progress.

3 Concurrent Control

While the features and properties of CCT algorithm provide great potential for fast parallel simulations, it is also crucial to performance to have well designed and efficient concurrent control mechanisms, which are indispensable in parallel execution.

The fine event granularity of network simulations dictates that conventional concurrent control methods could not provide the required efficiency. In This section, we present a number of mechanisms which provide lock-free synchronization in different situations between processors that are executing in parallel. These mechanisms are based on shared-memory multiple processor platforms and have been tested extensively in TasKit kernel. They are both effective and efficient. For limited space, we only give design principles.

3.1 Channel Message Queue Access

When the sender LP and the receiver LP of a channel execute in parallel, simultaneous access to the channel occur. A channel contains a message queue. Only two LPs shared the queue. The sender LP adds messages at the Head of the queue and the receiver LP removes messages from the Tail (see figure 5). The key idea for a lock-free scheme is to prevent the receiver from removing the message which the sender is still working on. For this purpose, the two pointers, Head and Tail, should never point to nil. As shown in the figure, Initially the queue is empty and both Head and Tail point to a dummy message.

When messages are added and removed, Tail points to a message that was last removed logically (to be placed into receiver's event list), but still physically remains in the queue. A separate link pointer field in the message buffer, other than Next, must be used for the event list. When the next message is removed
3.2 Processor Coordination

A ready task contains a number of events that can be processed safely without the fear of causal errors with respect to events that will be generated in other tasks. With a concurrent channel access mechanism in place, processors can execute tasks without interfering with each other. Normally, a task can only be executed by a single processor as is the case with cluster tasks where no parallel "sub-units" can be readily exploited.

However, pipe-type tasks contain pipeline parallelism that can be easily exploited. A processor, say \( P_1 \), takes a pipe-task and begins executing the first LP, e.g. \( LP_1 \) in figure 4. When \( P_1 \) finishes with \( LP_1 \), one of the input channels of \( LP_1 \) will again be critical and this task could be scheduled again by another task. \( P_1 \) will move on to execute \( LP_2 \), and then to \( LP_3, LP_4 \) and so on. Another processor \( P_2 \) could subsequently pick up the same task and begin executing \( LP_1 \) before \( P_1 \) has completed executing in the task.

A mechanism is required to coordinate processors executing in the same pipe-task. Each LP has a flag used to indicate whether or not it is busy. A processor has to acquire permission by checking the flag before it starts executing an LP. If the LP is busy, the processor will spin and wait until the flag is cleared. Then the processor will go ahead with that LP, sets the LP's flag to busy and starts executing. After it finishes with current LP, it will try to acquire and set the flag of next LP (if there is one) before it clears the flag of current LP. This order of "acquire and set next flag" creates a continuous protection to prevent more than one processor from entering the same LP at the same time.

However, problem remains at the first LP in a pipe-task. If there is a racing condition — meaning more than one processor may try to enter the first LP at the same time, a busy flag with no lock protection will not work. Fortunately, the lock used to protect a task queue could also be used to eliminate racing condition at the entry of a pipe-task.

3.3 Coordination on Critical Channels

Critical channels are set and cleared by their receiver LP's. It is possible that in busy execution a sender LP may try to schedule a receiver LP and its task while the receiver is still in execution. In this case, the sender might not be able to see the critical channel that will eventually be set just because it runs a bit too faster. If this happens the task may never be scheduled again and simulation could terminate prematurely.

There are two solutions to this problem. One method is called spin-wait. The receiver LP clears its critical channel after setting busy flag and before processing events. The sender LP checks the receiver's busy flag after it has set the clock of the channel. If the flag is busy, the sender spins and waits for the flag to be cleared. It then checks the channel's status and, if critical, schedules the task the receiver LP belongs.

The disadvantage of spin-wait is that processors could spend too much time waiting in applications with light task load. In the second method, instead of spin-wait, the sender LP sets the clock of the channel and, if critical, schedules the receiver LP, or moves on without wait. The receiver LP checks, after execution, the clock value of the new critical channel. If the clock value has advanced from the safe-time the receiver schedules its own task.

3.4 Termination

Terminating a simulation gracefully on a pre-specified condition, e.g. when simulation time has advanced to a specified end-time, could be a non-trivial job. It requires global information and coordination between processors. In TasKit kernel, a global count, \( N_{busy} \), is used to signal the end of simulation. When a processor gets a task and changes its state from idle to busy, it increases \( N_{busy} \) by 1. When it finishes with a task and finds no new task in the task queue, it decrements \( N_{busy} \) by 1 and becomes idle or, if \( N_{busy} \) becomes zero, quits the simulation. Updating \( N_{busy} \) must be locked, but locking alone is not sufficient to a safe termination. It is possible that one processor changing to idle finds zero \( N_{busy} \) and, at the same time, another processor switching from idle to busy is about to increment \( N_{busy} \). This will also result in a pre-mature termination.

For the centralized task queue scheme, updating \( N_{busy} \) is carried out together with task fetching using the same lock that is used to protect access to the task queue. For multiple task queues, \( N_{busy} \) signal alone cannot provide correct termination. Complicated mechanism would be required for a strictly correct \( N_{busy} \) at any given time. TasKit adopts a
practical approach that provides graceful termination without resorting to a complex procedure and the overhead associated with it.

Here is an example of how $N_{busy}$ could become false. Suppose processor $P_1$ has scheduled a task into task queue 2 where the task belongs, then finds its own task queue 1 empty, so it decrements $N_{busy}$ and goes idle. This could happen before processor $P_2$, the owner of queue 2, has taken the newly scheduled task from queue 2 and goes from idle to busy by incrementing $N_{busy}$. As a result, $P_1$ will see zero $N_{busy}$, therefore assume it is time to terminate and quit the simulation.

![Figure 6: Processor state transition diagram](image)

A simple yet effective solution, called delayed idling, can solve the problem. Another state standby is added between busy and idling states as shown in figure 6. Standby is an idling state when the processor still keeps its share of $N_{busy}$ count. The maximum duration of standby can be set long enough to cover the longest possible gap in updating $N_{busy}$ as exemplified above. The penalty is that termination will be delayed for the maximum duration the last processor has to stand by for, which can be ignored compared to the simulation time. A byproduct of delayed idling is reduced number of $N_{busy}$ locking. It is even possible that no $N_{busy}$ updating hence locking are necessary if each processor simply stands by long enough so that there must be no work to do, then simply quits.

4 Conclusions

This paper has covered the features and properties of a new conservative synchronization algorithm, Critical Channel Traversing, which works for both sequential and parallel discrete event simulation.

The CCT algorithm incorporates scheduling decisions into a conservative causal event ordering algorithm. It is capable of scheduling the LPs that will make most progress and providing coarse grain parallelism in a fine grain system. The algorithm also supports good cache behavior and automatic load balancing. Together with a set of lock-free concurrent control mechanisms, the new PDES kernel has the ability to support high performance communication network simulations.

References