Proceedings of the 1989 Winter Simulation Conference
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

EXPERIENCE WITH CONCURRENT SIMULATION

Douglas W. Jones
Chien-Chun Chou
Debra Renk
Steven C. Bruell
Department of Computer Science
University of Iowa
Iowa City, Iowa 52242, U.S.A.

ABSTRACT

We have successfully implemented a concurrent simulator and we report initial speedup measurements for this technique. Our algorithm requires a shared-memory multiprocessor and is appropriate to any discrete-event simulation model. In addition, our algorithm places no constraints on minimum service times or the presence of cycles in the simulation model. Our experimental simulator runs on an Encore Multimax computer and is applicable to closed product-form queueing networks. We have observed speedups of 2.3 for a central-server queueing model using our simulator.

1. INTRODUCTION

We have successfully implemented the concurrent simulation paradigm first introduced by Jones (1986b). Concurrent simulation allows parallel execution of discrete-event simulation models on shared-memory multiprocessors. It is insensitive to the topology of the model, and it can be applied to any discrete-event model, even cyclic models with no lower bound on the delay around a cycle.

Concurrent simulation is event-centered and it relies on a new shared abstract data type, the *concurrent pending-event set*. We have implemented this type using both a linear list and a skew heap. We have observed speed-ups of 2.3 using 4 processors on a 5 node central server queueing network model with our linear-list implementation.

In the following sections, we describe our approach, concurrent simulation, and contrast it with other approaches to discrete-event simulation (Section 2), our implementation (Section 3), and our initial empirical results (Section 4). Following this, we comment on the issue of correctness of parallel simulations (Section 5) and draw conclusions (Section 6).

2. DISCRETE-EVENT SIMULATION

A physical system is subject to discrete-event simulation if its abstract state can be described in terms of a set of variables that change discontinuously at discrete instants in time. These variables are the state variables of the discrete-event model, and the instants at which they change are the events in the model. At any point during the simulation of such a model, there is a pending-event set consisting of events that will occur at specific future times as consequences of events that have already occurred.

Events in the pending-event set can be described algorithmically. As input, they take the values of the state variables of the model prior to the event, and they deliver, as output, updated values of the state variables and new pending events. In an event-centered simulation model, pending events are represented by records containing parameters to one of a fixed set of event service routines.

In addition to the event-centered view of discrete-event simulation, there are two other views, the process-centered view and the activity-centered view (Overstreet, 1982, 1986). In both of these views, pending events are represented by records of suspended processes. In the process-centered view, many state variables are represented by local variables of cyclic processes, while in the activity-centered view, all processes are non-cyclic and are created in response to conditions detected in global state variables.

To run a discrete-event simulation model, one must provide an initial set of pending events and initial values for the state variables. Deterministic discrete-event models are said to be correct if the actual events in the simulated model occur at the same simulated times as the corresponding events in the physical system. Correctness of nondeterministic models is harder to define, but a reasonable criterion is that the distribution of observed outcomes after a number of runs starting from the same initial state match the distribution of outcomes observed for the physical system.

2.1. Sequential Simulation

Any discrete-event simulator must assure that events are simulated in the right order. The simplest way to do this is to simulate events sequentially in the same order they occur in the physical system. This order can be obtained by implementing the pending-event set as a priority queue, ordered by the simulated time at which the events are to occur. The simulation program repeatedly removes the head event from the queue and simulates it, possibly adding new events to the queue each time.

A significant amount of research has gone into finding fast implementations of the pending-event set for sequential simulation (Kingston, 1984, 1985; Jones et al, 1986), but further speedup is possible if events can be simulated in parallel. A parallel execution schedule for a deterministic discrete-event simulation run is correct if it simulates the same events at the same simulated times as would be obtained by a sequential schedule. Correctness is harder to define for the nondeterministic case, but asking for a similar distribution of outcomes after numerous runs is reasonable.

2.2. Distributed Simulation

All distributed simulation algorithms with which we are familiar are characterized by a process-centered view of discrete-event

simulation. No state variables are allowed except the local variables of logical processes in the simulation model. Logical processes each maintain a local record of the simulated time, and they communicate by passing messages, where each message contains, in addition to any other contents, the simulated time at which it is to be received (Chandy, 1981; Peacock, Wong and Manning, 1979a).

In effect, each logical process has its own pending-event set. This contains messages from other logical processes to which it may respond, and it contains messages to itself posted as a result of earlier events at that process.

A distributed simulation algorithm must prevent a logical process from processing received messages out of order. If the head message in the local message queue is to be processed at simulated time t, the logical process may not simulate it until it can be guaranteed that no new messages will be enqueued at times prior to t. Optimistic approaches to this have been proposed by Jefferson (1985); in these, logical processes are rolled back to a previous state if this constraint is violated. The problem with this approach is that it requires a large amount of memory to hold prior states of processes in case rollback is needed, and the problem of garbage collecting old states when they are no longer needed is not trivial.

Most solutions to the problem of synchronizing distributed simulations have been pessimistic; among these, Chandy and Misra's null-message algorithm (Chandy and Misra, 1979, 1981; Misra, 1986) and the link-time algorithm developed by Peacock, Wong and Manning (1979a, 1979b, 1979c) are perhaps the best known. Peacock, Wong and Manning (1979a) also described a blocking table algorithm that is, in a sense, optimal, and Lubachevsky (1988) has devised an interesting alternative for SIMD machines.

There are two primary problems with the pessimistic approaches. First, they are not universally applicable. They cannot handle cyclic models where a message sent out by one logical process could possibly circulate through a series of other logical processes and arrive back at the original sender at the time it was sent. Many systems are modelled using exponential service times, and because these have a minimum delay of zero, they must be modified for use on distributed simulators. A second problem is that, in addition to passing messages relevant to the model, most distributed simulators pass a significant number of additional synchronization messages.

2.3. Concurrent Simulation

In our approach to concurrent simulation, we take an event-centered view. There is no fixed binding between the processes in the simulator and any logical processes or other components of the simulation model. Simulation processes are servers, each of which repeatedly waits for a pending event to become available for simulation and then simulates it. The concurrent pending-event set is central to this approach. As with the conventional pending-event set, it organizes pending events in chronological order. In addition, it must prevent events from being removed for simulation until there is an assurance that no events will be scheduled at earlier times.

The fact that multiple processes may simulate events in parallel requires mutual exclusion for access to the state variables of the simulation model. In the simplest formulations, we assume that there is only one state variable, a record containing the entire state of the model. The result of this is the division of the process of simulating an event into a *state-variable phase*, during which the process has exclusive use of the state variables, and a (possibly empty) *event-scheduling* phase during which the event schedules other events.

The event-scheduling phases of simulation processes must not only insert events in the pending-event set, but it must help the event set determine which events are ready to be simulated. An event is ready to simulate if it is at the head of the pending-event set and if the process that scheduled it can guarantee that it will never schedule additional events ahead of it. As a result of these considerations, we identify the following operations as characterizing the concurrent pending-event set.

get-next – returns the head event from the pending-event set, possibly after waiting for it to be released; does not remove head event from set, but leaves it in place to ensure mutual exclusion during the state-variable phase.

insert – schedules a new pending event, but does not allow *get-next* to remove that event for simulation.

release – if applied to an event newly scheduled by *insert*, this operation allows *get-next* to return that event for simulation; if applied to the head event after that event was returned by *get-next*, this operation removes the event from the pending-event set.

The release operation is deliberately overloaded to simplify the specification of a simulation process by eliminating the need to make a special case of the end of the state-variable phase of each event. Jones (1986b) called the *get-next* and *insert* operations by the names *dequeue* and *enqueue*, respectively; we changed the name when we realized that *get-next* should not remove events from the pending-event set.

The concurrent pending-event set, the state variables of the simulation model, and the model itself are shared between all of the simulation processes. All other variables are local to each simulation process. In its most general form, the structure of a simulation process is outlined in Figure 1.

In this code, we make no assumptions about the values assigned to the time fields of each event. As a result, the event scheduling phase may overlap the state-variable phase arbitrarily. As a result, it is difficult to determine when it is safe to release an event. This difficulty leads to a direct statement of the conditions under which the *release* operation is appropriate The arguments that lead to this approach are detailed more fully by Jones (1986b).

This approach will achieve its maximum potential for parallelism if the event scheduling phase of each event schedules new events in chronological order. In this case, it will be legal to release each event immediately after the following event, and there is no need for additional computation to determine when the *release* operation is legal. In this case, we get the structure shown in Figure 2.

Here, the state-variable phase ends as soon as the first new event (if any) is scheduled, and new events are released for simulation as soon as the following event is scheduled. Ideally, no computation should occur in the state-variable phase other than update of state variables of the simulation model and the

```
var e1, e2 ... en: eventref;
repeat { body of each simulation process }
   get_next(e1) { enter state-variable phase };
   { body of state-variable phase }
   { start event scheduling phase }
   for i := 2 to n do begin
      insert(ei);
      for j := 1 to n do begin
         ok := false;
         { is at least one event inserted at a later time? }
         for k := 2 to n do begin
            if (ek.state ≠ free) and (ek.time ≥ ej.time)
            then ok := true;
         end;
         { and are all events at prior times inserted? }
         for k := 2 to n do begin
            if (ek.state = free) and (ek.time < ej.time)
            then ok := false;
         end;
         if ok then release(ej);
      end;
   end;
until simulation-completed;
```

Figure 1: Basic structure of a simulation process.

```
var e1, e2 ... en: eventref;
repeat { body of each simulation process }
   getnext(e1) { enter state-variable phase };
   { body of state-variable phase }
   { start event scheduling phase }
   for i = 2 to n do begin
      j := i - 1;
      { assert ei.time ≥ ej.time }
      insert(ei);
      release(ej);
   end;
   release(en);
until simulation-completed;
```

Figure 2: Modified structure of a simulation process.

determination of the time at which the first new event is to be scheduled.

Effective exploitation of concurrent simulation requires an event-set implementation that allows multiple processes to concurrently schedule new events. Furthermore, because the state-variable phases of all events execute sequentially, it is necessary to move as much computational work out of them as possible. Specifically, computation devoted to actually getting the next event should be outside of the state-variable phase if at all possible.

3. OUR IMPLEMENTATION

Our implementation of concurrent simulation proceeded by a systematic sequence of steps, starting with the abstract ideas outlined by Jones (1986b) and resulting in the development of efficient executable code. Before attempting an implementation, we divided the problem into two subproblems: the concurrent simulation algorithm itself and the concurrent pending-event set abstract data type needed to support the simulation.

3.1. Algebraic Specification

We began our implementation efforts with a formal algebraic definition of the *insert*, *get-next* and *release* operations. Algebraic methods have traditionally been applied to defining abstract data types in the context of sequential programs, and we had to modify this approach for use in the context of concurrent programming.

We augmented the algebraic rules applying to the operations on the concurrent pending-event set by adding rules that reduced illegal sequences of operations to an error state. For example, we introduced rules that stated that applying *get-next* to an event-set where the head element had not been released was an error. In effect, each of these added rules documents a synchronization constraint. The constraint corresponding to the above example is that the *get-next* operation must be blocked until *release* is applied to the head element.

Our formal definition helped us to understand that an event record may be in one of four states: *free*, *inserted*, *released*, and *current*. Events outside the event set are *free*. When an *insert* operation is applied to an event record, its state changes to *inserted*. The *release* operation changes the state of an *inserted* event to *released*, and the *get-next* operation changes the state of a *released* event to *current*. Finally, when the *release* operation is applied to a *current* event, the state changes to *free*. This cycle is illustrated in Figure 3.

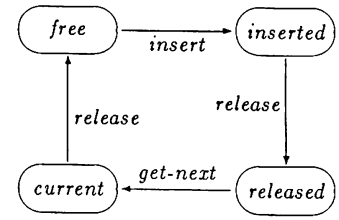


Figure 3: The states of an event record

It is worth noting that *get-next*, as specified here, is almost entirely a synchronization and state-change operation. The bulk of the work involved in managing the pending-event set is associated with *insert* and with *release* applied to the current event. There can never be more than one current event because the presence of a current event in the pending-event set will block additional *get-next* operations.

3.2. Linear-List Implementation

Our first implementation of the concurrent pending-event set was based on a trivial linear-list implementation. The purpose of this implementation was to obtain a correct implementation, allowing early experiments with the structure of concurrent simulation applications. Our only other performance goal was that our implementation allow operations initiated on the pending-event set by different simulation processes to take place in parallel.

The data structures required by the linear-list implementation are described in Figure 4. The event-set is implemented as a linear linked list of records, each containing the time of some pending event and other data needed to simulate that event. The final event-record in the list has an infinite time, or some time value larger than the time of any legitimate event encountered in the course of simulation. This eliminates the need for code to handle the empty list or the end of the list.

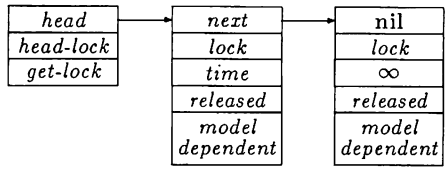


Figure 4: Linear-list event-set data structures

Locks protect *head*, the pointer to the head of the list, and *next*, the pointer in each list element to the following element. *Get-lock*, the second lock in the head of the list, is used to block *get-next* until the head event in the pending-event set has been released. This is locked by *get-next* and unlocked by any event-set operation that leaves the head of the event set in the *released* state.

The state of an event record is partly determined by the *released* field in each record. If this field is false, the record is *inserted*; if it is true, the record is either *released* or *current*. A correct simulation program will never attempt to release an *inserted* event more than once, so we take any attempt to release an already released event as an attempt to release the current event.

In the following code, we will assume that the operations *wait* and *signal* apply to locks. Logically, a lock is a binary semaphore, but the frequency with which this code makes requests for lock operations suggests using a spin-lock or busy waiting implementation. The code for *insert*, *get-next*, and *release* is given in Pascal in Figures 5, 6 and 7.

Note in this code that no operation on the concurrent pending-event set holds more than a small number of locks at a time. The *insert* operation holds locks in a clearly defined pattern we call the bubble of mutual exclusion. This bubble enters the list at the head and moves down the list until it encloses the point where the next insertion will be made.

```
procedure insert(e: event_ref);
var i, j: event_ref;
begin
    assert e↑.lock := unlocked;
    assert e↑.released := false;
    wait(head_lock);
    if e↑.time < head↑.time then begin
        e↑.next := head;
        head := e;
        signal(head_lock);
    end else begin
        i := head;
        wait(i↑.lock);
        signal(head_lock);
        j := i↑.next;
        while e↑.time ≥ j↑.time do begin
            wait(j↑.lock);
            signal(i↑.lock);
            i := j;
            j := i↑.next;
        end;
        e↑.next := j;
        i↑.next := e;
        signal(i↑.lock);
    end;
end;
```

Figure 5: Code for *insert*.

```
procedure get_next(var e: event_ref);
begin
    lock(get_lock);
    e := head;
end;
```

Figure 6: Code for *get-next*.

```
procedure release(e: event_ref);
begin
    lock(head_lock);
    if e = head then begin
        if e↑.released then begin
            { it is release(current event) }
            lock(e↑.lock);
            head := e↑.next;
            if head↑.released then unlock(get_lock);
            unlock(head_lock);
        end else begin
            { it is release(inserted event) }
            e↑.released := true;
            unlock(get_lock);
            unlock(head_lock);
        end;
    end else begin
        e↑.released := true;
        unlock(head_lock);
    end;
end;
```

Figure 7: Code for *release*.

3.3. Improved Performance

The basic code presented here was sufficient to allow us to test concurrent simulation, but the performance was not very impressive. We have found two productive ways to improve the performance of this code. Both rest on the observation that the state-variable phase of the simulation is the bottleneck in concurrent simulation.

Combining Insert and Release: Our first improvement involves a change to the basic operations on the concurrent pending-event set. This was motivated by the observation that the end of the state-variable phase occurs when the current event is released. The sooner this release operation can be done, the sooner some other process can begin simulating the next event.

All of our applications of concurrent simulation have the structure shown in Figure 2. As a result, every call to *insert* is followed immediately by a call to *release*, and the last call to *insert* is followed by two calls to *release*.

As a result, concurrent simulation programs can be compactly expressed in terms of the operations *insert-release* and *insert-release-release*. The semantics of these are defined as follows:

Insert-release(e1, e2) has the same semantics as *insert(e1)* followed by *release(e2)*.

Insert-release-release(e1, e2) has the same semantics as the sequence *insert(e1)* followed by *release(e2)* followed by *release(e1)*.

These combined operations can improve performance because the release operation can be carried out during the insert operation. Specifically, once it has been determined that the newly inserted item will not be the head item in the pending-event set, a *release* of the head item becomes possible. Thus, it is possible to end the state-variable phase of an event before the first *insert* operation has been completed. In fact, as soon as it is determined that an *insert* does not apply to the head element, the following *release* can be safely applied to any element of the event set without changing the outcome of the simulation.

The code for *insert-release* can be constructed by moving most of the code from Figure 7 into Figure 5 immediately after the else clause. The code for *insert-release-release* differs from *insert-release* in that it inserts the new event record with the *released* field set to *true* and it must check to see if *get-lock* should be unlocked.

Moving Work out of the State-Variable Phase: In examining the code for our initial applications of concurrent simulation, we found that there were a number of expensive operations being carried out in the state-variable phases of many of our event service routines. Among these were random number generation, allocation of event records, and accumulation of statistics. It turned out that many of these can be moved out of the state-variable phase.

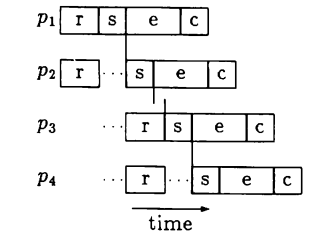
Our original focus was on random number generation. Generating uniformly distributed pseudo-random numbers is relatively easy, but the computations needed to convert from the uniform distribution to other distributions can be time consuming. One solution we considered was to dedicate a process to random number generation; this process would feed a stream of random numbers to the simulation processes.

Our final solution was to add a *resource acquisition phase* to the cycle of each simulation process. This phase takes place before the simulation process calls *get-next* to begin the simulation of an event. During the resource acquisition phase, the process acquires any consumable resources it might need to simulate the next event. This includes generating a collection of random numbers and allocating a collection of free pending-event records.

For an example of the resources that may be acquired in the resource acquisition phase, consider the simulation of a product-form queueing network. Each event in such a simulation signifies the end of service at some server. The simulation of an event may involve scheduling as many as two new events, one at the current server, and one at the next server that services the current customer. Each of these events requires an event record and a simulated time, so the simulator can acquire two event records and two appropriately distributed random numbers during the resource acquisition phase. If some of these resources are not used during the simulation of the next event, they may be saved for use with future events.

We also observed that many of the computations made during the state-variable phase had to do with integration, for example, integrating to determine average queue lengths in a queueing network. These computations could be done in any order, without reference to the synchronization constraints of concurrent simulation, and we found we could move them to a *cleanup phase* after the event-scheduling phase. Other operations appropriate to the cleanup phase include deallocating resources freed by the simulation of an event and deallocating the event-record itself.

The pipelined execution of a concurrent simulation augmented by the addition of these new phases is illustrated in Figure 8. Processes p_1 and p_2 in Figure 8 start immediately, but p_1 enters it's state-variable phase first, thus forcing p_2 to wait. The start of p_3 and p_4 has been delayed sufficiently that no process is ready to enter its state-variable phase when p_2 is done.



r – resource-acquisition phase
s – state-variable phase
e – event-scheduling phase
c – cleanup phase

Figure 8: Pipelined view of concurrent simulation

3.4. Heap Implementations

In parallel with our development of improved versions of the linear-list implementation of the concurrent pending-event set, we have begun to explore the use of more interesting data structures. Specifically, we have been working with the concurrent skew-heaps developed by Jones (1989) and the concurrent heaps developed by Rao and Kumar (1988). Preliminary indications are that these data structures can fully support both the basic operations *insert*, *release* and *get-next* and also the efficient combined operations *insert-release* and *insert-release-release*.

In the concurrent simulation environment with n events in the pending-event set, our linear list implementation takes O(1) time for *get-next* and *release* and O(n) time for *insert*, but *insert* holds exclusive use of the event set for only O(1) time. Thus, we expect to be able to make effective use of O(n) processors in parallel.

Skew-heaps and concurrent heaps should allow *get-next* and *release* (applied to enqueued events) in O(1) time, but *insert* and *release* (applied to the current event) should take O(log n) time, again holding exclusive use of the event set for only O(1) time. This should lead to effective use of O(log n) processors.

Finally, we are interested in experimenting with splay trees (Jones, 1986a) and with calendar queues (Brown, 1988). The former may be faster than the heap implementations, with O(1) expected times for *get-next* and *release* and O(log n) expected times for *insert*. The latter has worst-case times of O(n) for some operations but may well have average case times of O(1) for all operations.

3.5. Debugging Support

We feel that we were able to code and debug our concurrent simulator so easily because we first developed the algebraic specification described previously. Moving from this specification to working code was straightforward, but we found that it was hard to visualize what the resulting code was doing.

We implemented a program to graphically display an execution trace of the concurrent simulator to help us visualize what was actually happening. We modified one of our concurrent simulators to produce a trace indicating what each process was doing at each instant, and then used this as input to our display program.

The display program displays each record in the concurrent pending-event set as a small rectangle when the element has not been released. The *release* operation enlarges the rectangle, and the color of the rectangle is used to indicate which process, if any, has claimed the lock on that record. We did not uncover any errors using this tool, but we feel that the effort expended was justified because it gave us a clear picture of how our algorithm works.

4. EMPIRICAL RESULTS

We have decided to run our initial tests of concurrent simulation on discrete-event models of closed queueing networks. We have chosen this area because queueing networks are relatively easy to simulate and because the possibility of analytical solution allows an independent check on the results of our simulations.

All of the queueing networks we have explored were closed, and thus contained cycles, and all of the servers had exponentially distributed service times. As such, our models have no minimum service time; in the presence of cycles, this implies our models are not suited for many approaches to distributed simulation (Reed, Malony and McCredie, 1988).

It should be noted that no event in this type of queueing network can ever schedule more than two new events. As such, the event-scheduling phase of a concurrent simulator for such a model will be very modest, leading to only limited possibilities for parallelism. As a result, we feel that our results understate the potential speedup to be obtained from concurrent simulation.

All of our speedup figures are reported by comparison with a simulation program written for a uniprocessor. This sequential program is based on the code in Figures 7.7 and 7.11 of Sauer and Chandy (1981), which uses a linear list to represent the pending-event set. For small models, where the maximum size of the pending-event set is less than about 20, linear list implementations should be optimal. Had we measured speedups relative to the concurrent simulation algorithm running with only one process, we would have reported larger speedups.

All of our code is written in Pascal and tested on Encore Multimax computers running the UMAX 4.2 operating system, a variant of BSD 4.2 UNIX. (The general architecture of this class of machine was described by Bell (1985).) We use the *spin-lock* primitives provided by Encore to implement the locks required by our algorithm, and we use the UNIX *fork* primitive for creating multiple processes. The Encore machine for which we have presented results has 56 Mbytes of memory shared by 18 NS32332 processors with Witek floating point accelerators.

4.1. Initial Experiments

Initially, we built a special purpose queueing network simulator that could simulate queueing networks consisting of a single cycle of single-server queues. Such a network with three servers is illustrated in Figure 9. All stations had a mean service time of one, and the model was parameterized by the number of servers and the population.

Preliminary measurements of a 40 server cyclic network with 80 customers showed a speedup of 1.3 using two processes. This measurement was made before we moved work out of the state-variable phase into the resource acquisition phase and the cleanup phase.

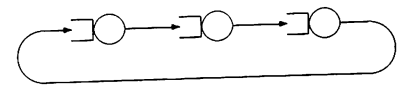


Figure 9: A three station cyclic queueing network.

4.2. A General Program

We found that it was very easy to merge our concurrent simulator for cyclic queueing networks with an existing general purpose queueing network package developed by Bruell, Balbo and Ghanta (1984). This package allows arbitrary closed queueing networks to be described and solved analytically or by simulation.

We have used the resulting general purpose simulator to run experiments with the queueing model previously used by Wagner, Lazowska and Bershad (1988) and Reed, Malony and McCredie (1988); this is shown in Figure 10. The numbers shown inside the servers are the mean service times and those shown on the arcs out of the fork server are transition probabilities. For our initial tests of this network, we used 40 customers.

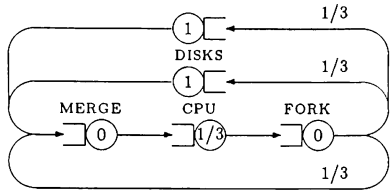


Figure 10: A five node central server model.

We chose this model because empirical results suggest very small speedups, never better than 1.25, using a number of approaches to distributed simulation (Reed, Malony and McCredie, 1988), and there is a theoretical proof that no conservative distributed simulation algorithm can excede speedups of 3.67 on this model due to Wagner, Lazowska and Bershad (1988). In addition, Wagner and Lazowska (1988) report speedups of about 2.8 on this model using "highly optomized simulation implementations".

As shown in Figure 11, we have achieved a speedup on this model of just over 2.3, using 3 or 4 processors, measured relative to clean uniprocessor code. We ran the simulation for 10,000 simulated time units, which comes to about 105,000 events per run. The maximum pending-event set size for this model is 5 events, one per server when that server is busy. The fork and merge servers each have service times of zero, so we expect pending-event set sizes on the order of 3.

Both Reed, Malony and McCredie (1988) and Wagner Lazowska and Bershad (1988) measured their speedups relative to their distributed simulation code running on a single processor. If we measure our speedups this way, we get a speedup of just under 2.8, almost identical to the speedup reported by Wagner Lazowska and Bershad (1988).

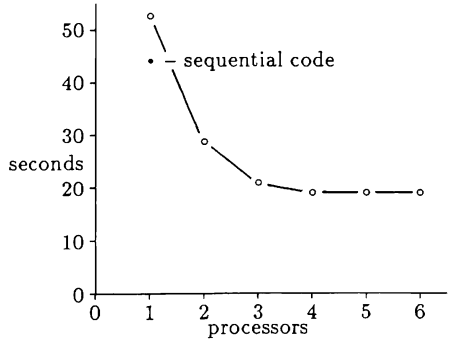


Figure 11: Run-times for the five node central server model.

5. CORRECTNESS ISSUES

If we define a concurrent simulation as being correct if it produces the same results as a conventional sequential simulation for the same model, our concurrent simulation algorithm cannot be considered to be correct! The problem does not lie in our algorithm as much as it lies in the conventional notion of correctness.

One reason for this lies in the generation of random numbers. In a conventional simulator, there is usually a single stream of pseudo-random numbers serving the entire simulator. In our concurrent simulators, each simulation process uses its own stream of pseudo-random numbers. This avoids contention for a single random-number generator but changes the outcome of the simulation.

Even if we shared the random number generator, we would get a different outcome once we separated the state-variable phase of the simulation from the resource acquisition phase. This is because resources need not be acquired in the order they are actually used. An important result of this is that two runs of the same simulation model with the same random number seeds will not necessarily give the same output because interrupts or other outside events may delay processors at random times, causing them to permute the random number stream in different ways.

Use of concurrent heaps or skew heaps will complicate the situation even more because these data structures are unstable. Unlike a linear list, these data structures do not guarantee that events scheduled at the same simulated time will be simulated in the order they are inserted.

As a result of such considerations, we must content ourselves with considering a simulation to be correct if the output it produces is statistically reasonable. In fact, this should not be a surprise, this is how we judge the correctness of sequential simulators, but we usually introduce a degree of artificial stability into sequential simulators to simplify debugging.

6. CONCLUSIONS

We conjecture that we will achieve the greatest speed-up when using concurrent simulation on models in which there is a wide variance in the number of new events scheduled by an event. In these cases the event scheduling phases of some events will be quite long and may effectively utilize a simulation processor for some time while other events are being simulated on other processors. If the variance is small (for example, if each event schedules exactly one new event), the only speed-up we expect will be the result of being able to release the current event before completing the scheduling of the next event.

Unlike distributed simulation, we do not expect speed-up to depend on the uniformity of the distribution of activity over the simulation model. Thus, if the pending-event set is large and there is a wide variance in event scheduling, then we expect good speed-up even in models for which distributed simulation would perform poorly.

We must emphasize that the speedups we have measured at this point are only preliminary. We do not believe that we have fully exploited the potential of concurrent simulation, and we have put only minimal efforts into code tuning and other refinements.

A natural idea to pursue is the combination of the best features of distributed and concurrent simulation. In the distributed simulation algorithm developed by Peacock, Wong, and Manning (1979a), each communication link between logical processes had an associated link-time. This is the simulation time before which the sending process guarantees that it will never send a message. The receiving process is free to simulate any event at simulated times less than all incoming link times. It may be possible to implement link times in a mixed concurrent/distributed model by associating a dummy event with each link; this event would never be released and would have as its time the link time of that link.

ACKNOWLEDGEMENTS

Certain contributions to this project are the results of individual efforts and should be given credit as such: Jones proposed the use of *insert-release*. Renk did the initial implementations of various versions of the concurrent pending event set. Bruell proposed the early release in the combined *insert-release* operation. Bruell and Chou proposed the resource acquisition and cleanup phases and performed most of the empirical tests.

Renk was supported throughout this work by an IBM graduate research fellowship.

REFERENCES

Bell, C. G. (1985) Multis: A New Class of Multiprocessor Computers. *Science, 231,* 462-467.

Brown, R. (1988) Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *Communications of the ACM, 31,* 1220-1227.

Bruell, S. C., Balbo, G., Ghanta S., and Afshari, P. V. (1984) A Mean Value Analysis Based Package for the Solution of Product-Form Queueing Network Models. *International Conference on Modelling Techniques and Tools for Performance Analysis,* Paris, France.

Chandy, K. M., and Misra J. (1979) Distributed Simulation: Case Study. *IEEE Transactions on Software Engineering, SE-5,* 440-452.

Chandy, K. M., and Misra J. (1981) Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM, 24,* 198-206.

Jefferson, D. R. (1985) Virtual Time. *ACM Transactions on Programming Languages and Systems, 7,* 404-425.

Jones, D. W. (1986a) An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communications of the ACM, 29,* 300-311.

Jones, D. W. (1986b) Concurrent Simulation: An Alternative to Distributed Simulation. *Proceedings of the 1986 Winter Simulation Conference,* Arlington, Virginia, 417-423.

Jones, D. W., Henriksen, J. O., Pegden, C. D., et al. (1986) Implementations of Time (Panel). *Proceedings of the 1986 Winter Simulation Conference,* Arlington, Virginia 409-416.

Jones, D. W. (1989) Concurrent Operations on Priority Queues. *Communications of the ACM, 32,* 132-137.

Kingston, J. H. (1984) Analysis of Algorithms for the Simulation Event List. PhD Thesis, Basser Department of Computer Science, University of Sydney.

Kingston, J. H. (1985) Analysis of Tree Algorithms for the Simulation Event List. *Acta Informatica, 22,* 15-33.

Lubachevsky, B. D. (1988) Efficient Distributed Event-driven Simulations of Multiple-loop Networks. *Proceeding of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* 12-21.

Misra, J. (1986) Distributed Discrete-event Simulation, *Computing Surveys, 18,* 39-65.

Overstreet, C. M. (1982) Model Specification and Analysis for Discrete Event Simulation. PhD Thesis, Computer Science Department, Virginia Polytechnic Institute, Blacksburg, Virginia.

Overstreet, C. M. (1986) World View Based Discrete Event Model Simplification. *Modelling and Simulation Methodology in the Artificial Intelligence Era.* North Holland, Amsterdam.

Peacock, J. K., Wong, J. W., and Manning, E. G. (1979a) Distributed Simulation Using a Network of Processors. *Computer Networks, 3,* 44-56.

Peacock, J. K., Wong, J. W., and Manning, E. G. (1979b) A Distributed Approach to Queueing Network Simulation. *1979 Winter Simulation Conference,* IEEE, 399-406.

Peacock, J. K., Wong, J. W. and Manning, E. G. (1979c) A Distributed Approach to Queueing Network Simulation. *Proceedings 4th Berkeley Conference on Distributed Data Management and Computer Networks,* Berkeley, CA, 237-259.

Rao, V. N., and Kumar, V. (1988) Concurrent Access of Priority Queues. *IEEE Transactions on Computers, 37,* 1657-1665.

Reed, D. A., Malony, A. D., and McCredie, B. D. (1988) Parallel Discrete Event Simulation Using Shared Memory. *IEEE Transactions on Software Engineering, 14,* 541-553.

Sauer, C. H., and Chandy K. M. (1981) *Computer Systems Performance Modelling.* Prentice-Hall, Englewood Cliffs, NJ.

Wagner, D. B., Lazowska, E. D., and Bershad, B. N. (1988) Techniques for Efficient Shared-Memory Parallel Simulation. Technical Report 88-04-05, Department of Computer Science, University of Washington, Seattle.

Wagner, D. B., and Lazowska, E. D., (1988) Parallel Simulation of Queueing Networks: Limitations and Potentials. Technical Report 88-09-05, Department of Computer Science, University of Washington, Seattle.

AUTHORS' BIOGRAPHIES

DOUGLAS W. JONES is an associate professor of computer science at the University of Iowa. He received his BS in physics from Carnegie-Mellon University in 1973 and his MS and PhD degrees in Computer Science from the University of Illinois in 1976 and 1980, respectively. In addition to work in parallel simulation, he has developed a VLSI description and simulation system, and he has done work in the area of algorithms, most recently, in the area of data compression.

Douglas W. Jones
Department of Computer Science
University of Iowa
Iowa City, IA 52242, U.S.A.
(319) 335-0740
Internet: jones@herky.cs.uiowa.edu

CHIEN-CHUN CHOU is a PhD student in computer science at the University of Iowa. He received his BS in computer science from Chung-Yuan Christian University (Taiwan) in 1983.

Chien-Chun Chou
Department of Computer Science
University of Iowa
Iowa City, IA 52242, U.S.A.
Internet: chou@herky.cs.uiowa.edu

DEBRA RENK is a PhD student in computer science at the University of Iowa. She received her BA in computer science from the Iowa in 1982.

Debra Renk
Department of Computer Science
University of Iowa
Iowa City, IA 52242, U.S.A.
Internet: renk@herky.cs.uiowa.edu

STEVEN C. BRUELL is an associate professor of computer science at the University of Iowa. He received his BS with honors from the University of Texas at Austin in 1973 and his MS and PhD degrees from Purdue University in 1975 and 1978, respectively. In addition to work in parallel simulation, he has worked in the areas of generalized stochastic Petri nets and queueing models applied to computer system performance modelling. He has also co-authored three books.

Steven C. Bruell
Department of Computer Science
University of Iowa
Iowa City, IA 52242, U.S.A.
(319) 335-0734
Internet: bruell@herky.cs.uiowa.edu