PROC IASTED Int. Conf. on Modeling Simulation and Optimization
1992
ISBN:0-88986-197-8 Under Number 242-019

A Conservative Distributed Concurrent Discrete-Event Simulator on a Shared Memory

Herbert R. Hoeger
Simulation and Modeling Center
Facultad de Ingenieria
Universidad de Los Andes
Mérida, Estado Mérida 5101
VENEZUELA
E-mail: hhoeger@ing.ula.ve

Douglas W. Jones
Department of Computer Science
University of Iowa
Iowa City, Iowa 52242
U.S.A.
E-mail: jones@cs.uiowa.edu

1. ABSTRACT

There have been many attempts to apply parallel computers to discrete-event simulation. These may be divided into two main approaches, distributed simulation and concurrent simulation.

Distributed simulation partitions the simulation model into components that can be executed on different processors. It usually follows a process-centered view of discrete-event simulation.

In concurrent simulation the simulation of events is divided in phases and processed in a pipelined manner. It is particularly appropriate for shared-memory systems and it takes an event-centered view of parallel simulation.

This paper focuses on the integration of these two approaches, resulting in a distributed simulator with concurrency added to each model component. To do so, we used a shared-memory environment and unified both approaches to an event-centered view.

Keywords: distributed concurrent discrete-event simulation.

2. DISCRETE-EVENT SIMULATION

To simulate means to create a model that approximates some aspect of a real-world system and that can be used to generate artificial histories of the system, so as to allow us to predict some aspect of the behavior of the system. A collection of variables, the state variables, are used to describe the system at any time. The state of a system is characterized by the values that the state variables have at an instant in time; in discrete-event simulation, the state variables change only at discrete points in simulated time, the events.

The occurrence of events drives the simulation. A list of pending events, the pending-event set, contains all the events that have been scheduled but not yet simulated. Simulating an event may involve changing the state variables and scheduling new events [1,2,3]. The simulation mechanism must guarantee

causality; i.e., that all events are simulated after all events on which their outcome depends.

2.1. Conservative Distributed Simulation

Distributed simulation is based on a spatial decomposition and partitions the simulation model into loosely coupled components that are executed on different processors, and that communicate by sending each other messages carrying events with timestamps or event messages. There are two main categories of distributed simulators: those that use conservative mechanisms and those that use optimistic mechanisms.

Conservative mechanisms do not allow a logical process to process an event message until it is certain that no other event message with a smaller timestamp can be scheduled. Optimistic mechanisms allow each logical process to continue processing event messages but, if they detect a causality error they are able to recover from their effects.

Our work is based on the conservative approach and therefore we will describe this mechanism with more care. For more details on all of these mechanisms see [3,4,5,6].

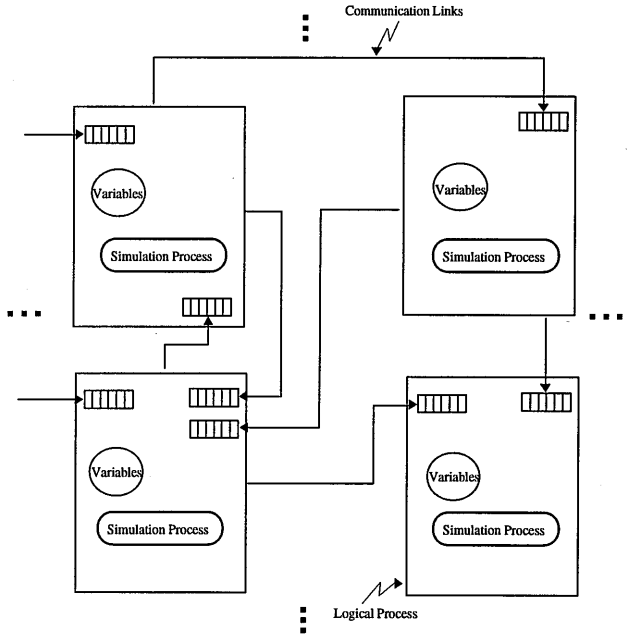

Figure 1. Conservative Distributed Simulator.

Some of the first conservative distributed simulation algorithms were developed by Chandy and Misra [4], and Bryant [7]; the best known conservative distributed algorithms follow a process-centered view of discrete-event simulation where the system being modeled is viewed as a number of interacting physical processes [8]. The simulator is a set of logical processes (see Figure 1), one for each physical process, with a

corresponding simulation process in charge of executing the simulation algorithm for that logical process. Interactions in the physical system are modeled by event messages sent along communication links. The set of state variables has to be partitioned so that each logical process has all the state variables it requires to run [3]. The simulation progresses asynchronously.

Distributed simulators offer the potential for speed-ups which scale with the model size because each logical process may potentially run on a different processor. A drawback of these simulators is that bottlenecks in the system being simulated become bottlenecks in the simulation and limit the possible speed-ups [9].

2.2. Concurrent Simulation

Concurrent simulation, an alternative to distributed simulation presented by Jones [8], relies on a temporal decomposition. The simulation of each event is divided into a state-variable phase and an event-scheduling phase. There are no bindings between events and the processes that perform the simulation; instead, these processes are just servers that repeatedly wait for events to be available and simulate them (see Figure 2). Given that many events may be simulated in parallel, access to the state variables requires mutual exclusion. The simplest approach to this is to require that at most one event be in its state-variable phase at any given instant, yet many events may be in their scheduling-phases simultaneously. A concurrent pending-event set organizes events in chronological order. No event is allowed to be removed from the set until there is assurance that no events will be scheduled earlier. The concurrent pending event-set, the state variables, and the simulation model are shared among the simulation processes [10].

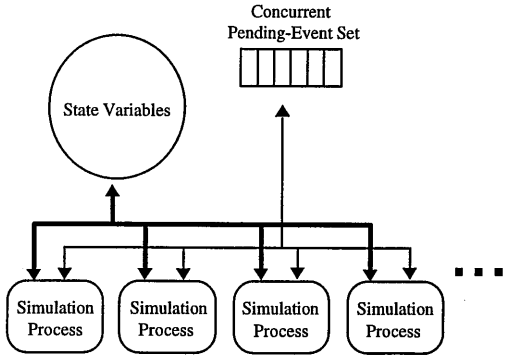

Figure 2. Concurrent Simulator.

Due to the pipelined scheme followed by concurrent simulators, they do not scale with the size of the model. At some point the added resource contention caused by extra processes will offset any potential speed-up. However, concurrent simulators inherently

balance the computational load over the available processors and are insensitive to bottlenecks in the model being simulated; the components with more load will have more events in the concurrent pending-event set and therefore will get more processing time than other components with less events in the event set.

3. THE INTEGRATED CONSERVATIVE DISTRIBUTED CONCURRENT SIMULATOR

We began our work with a conservative distributed simulator written by Chou [11], based on the algorithm developed by Chandy and Misra [4], and for each logical process we fused the queues associated with the communication links into a single concurrent pending event-set. We also allow model components to be grouped into what we called a *cluster*, and more than one simulation processes may be assigned to a logical process (see Figure 3).

This resulted in an *integrated conservative distributed concurrent discrete-event simulator* that runs faster than any of the previous conservative distributed simulators that Chou investigated, and depending on the topology of the model being simulated, runs faster than Chou's concurrent simulator.

This new simulator should scale with the model size like the distributed simulators, and should reduce the impact of bottlenecks in the model by attaching more simulation processes to them.

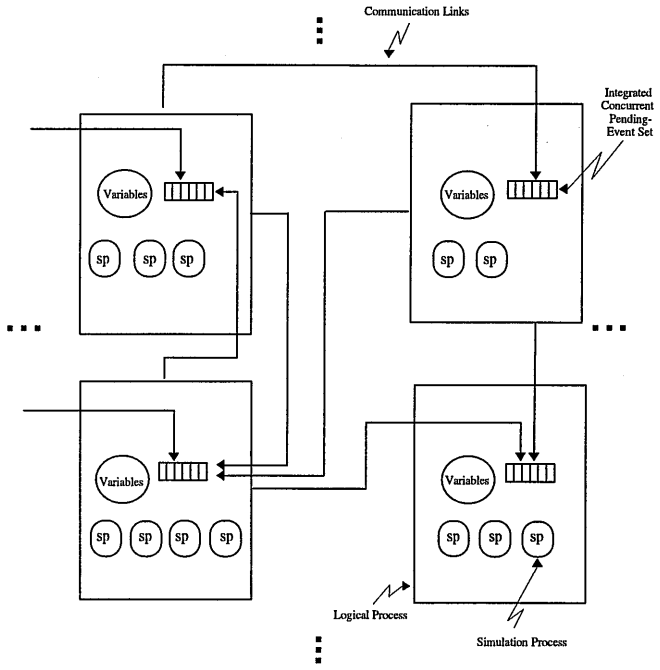

Figure 3. Integrated Distributed Concurrent Simulator.

Although, our new simulator operates on closed queueing network models, none of the underlying mechanisms are specific to

this application domain. A more complete presentation of this work is given in [12].

3.1. Changes to the Traditional View of Distributed Simulation

We fused the main data structure of conservative distributed simulation, the communication links, with the main data structure of concurrent simulation, the concurrent pending-event set. The result is a single integrated concurrent pending-event set per logical process in the model. This is only possible in a shared memory environment, and in order to do so we had to reformulate the description of the traditional conservative distributed simulation from a process-centered view to an event-centered view.

In the process-centered view the logical processes communicate by sending event messages trough the communication links. Each logical process is responsible processing the received messages in the right order. In the event-centered view a logical process has the ability to directly schedule events in the integrated pending-event sets of other logical processes. Thus, the ordering activity has been transferred from the logical process that will process the event to the logical process that generates the event.

3.2. Clustering Model Components

If each logical process models a single model component and requires a separate CPU for efficient simulation, large models cannot be efficiently simulated on today's multiprocessors. To overcome this limitation, we must assign more than one model component per logical process.

We call a group of model components a *cluster*. In our simulator, each logical process models the behavior of a cluster and has a single integrated pending-event set which holds events concerning all of the model components of the cluster that logical process is modeling.

3.3. Synchronizing the Logical Processes

We use null events to synchronize logical processes; these correspond roughly to the null messages of Chandy and Misra's distributed algorithm. Each logical process maintains a null event notice permanently in the pending-event set of each logical process to which it may communicate. The number of null events remains constant during the simulation and are created at initialization time. When a simulation process advances its local clock, it informs downstream process by rescheduling all of its null events. When a null event is rescheduled, it is removed from its current location in the pending-event set and then placed according to a new timestamp. The timestamp of a null event is the time before which no new events will be scheduled by the logical process that owns the null event. Whenever a logical

process finds a null event at the head of its pending-event set, it blocks until that null event is rescheduled.

3.4. Attaching Multiple Simulation Processes to a Cluster

In order to allow more than one simulation process to handle the events from an pending-event set, we added a lock variable to our event-set data structure. This lock will prevent another simulation process from processing events from the integrated pending-event set until there is assurance that no events will be scheduled before the head-event by other simulation processes in the cluster.

4. RESULTS

All of the simulators we tested, Chou's sequential simulator, his conservative distributed simulator, his concurrent simulator, and our integrated conservative distributed concurrent simulator, were implemented using Pascal on an Encore Multimax with fourteen NS-32532 processors. The simulation processes were implemented using UNIX processes (created using the fork system call). All UNIX processes were created at the beginning of the simulation and destroyed at the end of it, so there are no short lifetime processes and there is no dynamic creation of processes.

4.1. Preliminary Results

In our preliminary tests, we used the same two queuing network models used by Chou to test the new simulator and compare it with earlier simulators. One, the Central-Server Model (Figure 4), first used in Reed, Malony, and McCredie [13], and then in Wagner and Lazowska [9], has five stations representing a Merge node, a CPU, a Fork node, and two Disks. The mean service times (m.s.t.) are 0, 1, 0, 3, and 3, seconds respectively. The second model is a Cyclic Queue (Figure 5) with six stations each with a mean service time of 2 seconds. All service times were exponentially distributed.

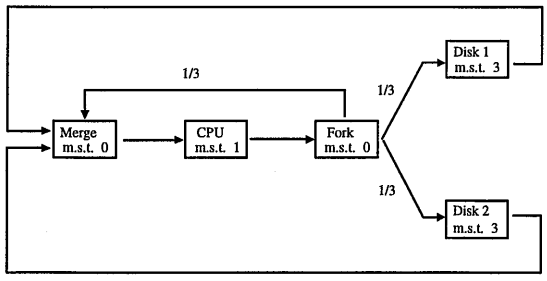

Figure 4. Central-Server Model with Explicit Merge and Fork Nodes.

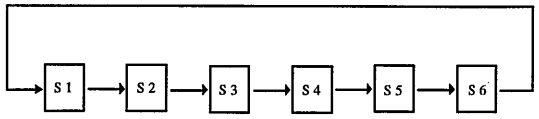

Figure 5. Cyclic Queuing Network with Six Stations.

For our new simulator there are basically no restrictions in the way in which the clusters can be built; any station may belong to any cluster. Thus, our simulator can be configured as a conservative distributed simulator with one station per cluster,

as a concurrent simulator with all the stations in one cluster, or anything in between.

By assigning more simulation processes to those stations with higher load, the new simulator yields better results than any of the previous simulators. For the cyclic-queue model, which is a well balanced model, both Chou's distributed simulator and our simulator with a plain distributed configuration yield a significant improvement over the concurrent simulator. This is because in the distributed simulators all the simulation processes have about the same load and the contention introduced by the concurrent simulator is avoided. Our simulator is significantly better than Chou's distributed simulator because of the overhead reduction due to the integrated pending-event sets.

4.2. Results with Larger Models

After we tested our new simulator with the simple models described in the previous section, we wanted to see how it behaves with larger models (with more stations) and compare it to the other simulators.

For models with a severe bottleneck, distribution gives a very poor performance as predicted by Wagner and Lazowska [9]. Even by assigning more processes to bottlenecks in our simulator, it is considerably slower than the sequential simulator. The concurrent simulator behaves well in these cases, because its algorithm implicitly takes care of balancing the load.

For well balanced models, our simulator always yields the best performance.

For models in between, the concurrent simulator and the integrated conservative distributed concurrent simulator show about the same performance.

5. CONCLUSIONS

We have shown that a conservative distributed concurrent simulator can be built, allowing model components to be grouped into clusters. Each logical process models a cluster, and we can attach more than one simulation process to a logical processes. This allows us to use our simulator as a concurrent simulator, as a distributed simulator, or as a mixture of both.

Our new simulator outperforms previous distributed simulators because of the overhead reduction that clustering introduces and because we can attach more processors to heavily loaded clusters. It does not perform well in every case though; specifically, if the model being simulated is poorly balanced, the concurrent simulator is better because its algorithm implicitly takes care of improper load balance in the model.

If the model is well balanced, however, our new simulator, outperforms the concurrent simulator. In some cases in which the model is unbalanced, our simulator performs better than the concurrent simulator when extra processors are assigned to heavily loaded clusters.

The speed-ups for distributed simulators can scale with the model size because each logical process can run on a different processor, so long as there are available processors in the system. Concurrent simulators do not scale with the size of the model. Due to the pipeline scheme they follow, processors can not be added indefinitely because after certain number, the contention of an extra processor will offset any potential speed-up. We hope our integrated conservative distributed concurrent simulator will scale like traditional distributed simulators while allowing load balancing like the concurrent simulator; further experiments will be needed to explore this.

REFERENCES

[1] J. Banks and J.S. Carson, *Discrete-event System Simulation*, Prentice-Hall, Englewood Cliffs, N.J., 1984.

[2] W.G. Bulgren, *Discrete System Simulation*, Prentice-Hall, Englewood Cliffs, N.J., 1982.

[3] R.M. Fujimoto, *Parallel Discrete Event Simulation*, Communications of the ACM, 33(10), 1990, 30-53.

[4] K.M. Chandy and J. Misra, *Distributed Simulation: A Case Study in Design and Verification of Distributed Programs*, IEEE Transactions on Software Engineering, SE-5(5), 1979, 440-452.

[5] D.R. Jefferson, *Virtual Time*, ACM Transactions on Programming Languages and Systems, 7(3), 1985, 404-425.

[6] R. Righter and J.C. Walrand, *Distributed Simulation of Discrete Event Systems*, Proceedings of the IEEE, 77(1), 1989, 99-113.

[7] R.E. Bryant, *Simulation of Packet Communications Architecture Computer Systems*, MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[8] D.W. Jones, *Concurrent Simulation: An Alternative to Distributed Simulation*, Proceedings of the Winter Simulation Conference, 1986, 417-423.

[9] D.B. Wagner and E.D. Lazowska, *Parallel Simulation of Queuing Networks: Limitations and Potential*, Proceedings of ACM SIGMETRICS and Performance, 17(1), 1989, 146-155.

[10] D.W. Jones, C-C. Chou, D. Renk, and S.C. Bruell, *Experience with Concurrent Simulation*, Proceedings of the Winter Simulation Conference, 1989, 756-764.

[11] C-C. Chou, *Parallel Simulation and its Performance Evaluation*, Ph.D. Thesis, Dept. of Computer Science, University of Iowa, Iowa City, Iowa, 1992.

[12] H.R. Hoeger, *Integration of Distributed and Concurrent Discrete-Event Simulation*, Ph.D. Thesis, Dept. of Computer Science, University of Iowa, Iowa City, Iowa, 1995.

[13] D.A. Reed, A.D. Malony, and B.D. McCredie, *Parallel Discrete Event Simulation Using Shared Memory*, IEEE Transactions on Software Engineering, 14(4), 1988, 541-553.