Parallel Discrete Event Simulation on Shared-Memory Multiprocessors *

Pavlos Konas
Pen-Chung Yew

Center for Supercomputing Research and Development,
305 Talbot Laboratory, 104 S. Wright Street,
Urbana, IL 61801.

Abstract

This paper describes the implementation and the performance study of three parallel discrete event simulation methods on a shared memory machine. These methods, which share a single user interface, include the Chandy-Misra paradigm with deadlock avoidance; the Time Warp approach with direct, aggressive, and lazy cancellation; and a hybrid approach, which exploits the parallelism available at each point in simulated time. In this study we also examine the impact of task-partitioning and of processor self-scheduling on the efficient implementation of the above methods.

Two kinds of systems are simulated: a synchronous multiprocessor machine and an asynchronous toroid network with FCFS server nodes. The performance of the implemented methods is discussed, and conclusions are drawn from the obtained results.

1 Introduction

As computer systems become more complex, it is increasingly difficult to use analytical models to describe their behavior and assess their performance [5]. Simulation not only assists us in describing and evaluating new systems, but also provides us with insight on the operation of existing ones. However, detailed simulations of computer systems require excessive amounts of CPU time, and their execution on sequential machines is prohibitively expensive. One way to meet the processing requirements of these applications is to execute them on a multiprocessor machine.

In this paper, we study the performance of three different approaches to the execution of a single sim-

*This work was supported in part by the National Science Foundation under Grant No. US NSF MIP 8410110 and NSF MIP F8-07775, the U.S. Department of Energy under Grant No. US DOE DE-FG02-85ER25001, NASA NCC 2-559, Digital Equipment Corporation and IBM Corporation.
eplished with a graphical user interface. Assisted by the graphical interface, the user can concentrate on the development of the actual simulation and execute it using any of the three methods. In this way, one does not waste time and effort in writing a new simulation kernel whenever wishes to execute a simulation in parallel.

In addition, the description of simulation components can be shared among different simulations, making the development of new simulations easier and faster.

Our concern in this paper is the performance of the PDES methods in the simulation of architectural designs. Section 2 describes the implemented methods and the optimizations we applied utilizing the shared memory of the Alliant. Sections 3, 4, and 5 examine two important issues in the efficient implementation of the PDES methods: task partitioning and processor self-scheduling. Section 6 presents the experiments we performed on the Alliant and the results we obtained. Finally, our conclusions as well as future directions are discussed in Section 7.

2 Parallel Simulation Methods

2.1 The Chandy-Misra and Bryant Method

The first conservative PDES algorithms were developed by Chandy and Misra [7] and Bryant [6]. In their method, a clock is associated with each input of a logical process and contains the time stamp of the first outstanding message on that input, or, in the case where the input is empty, the time stamp of the last message received over it. Each logical process repeatedly selects the input with the smallest clock value and, if it is not empty, receives the first message on that channel. If it is empty, the process blocks on every channel that has the smallest clock value. Although this protocol ensures that no causality error will ever occur, it may lead to deadlock [17]. Several mechanisms have been proposed to solve this problem [7, 17]. One approach is to have each process send a null message on each of its outgoing channels when it is about to block. This method solves the deadlock problem as long as there is no cycle of processes along which the sum of all time stamp increments is zero [7]. Since we use a shared memory machine, we do not actually send null messages. Instead, we associate a second clock with each channel that contains the time stamp of the last null message we would have sent along the channel. This optimization, also found in [9], avoids unnecessary enqueue and dequeue operations and leads to more efficient utilization of the shared memory.

2.2 The Time Warp Approach

The Time Warp mechanism, based on the Virtual Time paradigm [13], is the most well-known optimistic PDES protocol. In Time Warp, instead of determining when it is safe to process a message, a process is allowed to receive event messages continuously until no unprocessed messages remain on its input or until a message whose time stamp is smaller than the clock value of the process (a straggler message) arrives. The arrival of a straggler initiates the rollback procedure, which undoes the effects of all messages that have been processed prematurely (aggressive cancellation). A rollback consists of restoring the process to the appropriate state and canceling all messages produced during the pre-rollback computation. The erroneously produced messages, however, may have already been processed, so the corresponding receiver processes must also be rolled back. This procedure is repeated until all the effects of the erroneous computation are cancelled. This rollback wave is bounded, and the Time Warp mechanism always makes progress [12].

One optimization we can apply is to cancel only messages which are not reproduced as we perform the computation of rolled back events again (lazy cancellation [11]). This optimization reduces the rollback overhead, because in some case a straggler event changes only part or none of the computation of rolled back events, and subsequently some of the erroneously generated messages are still correct. Since the simulation is performed on a shared memory multiprocessor, another optimization that we applied is direct cancellation [8]. We augmented each event with a causality record which points to all the events generated by its processing. Rolling back the event requires deleting the messages pointed to be the causality record and rolling back the event lists where they reside (if necessary). This optimization reduces the event list operations and allows the cancellation procedure to proceed faster and be performed distributedly.

2.3 A Hybrid Approach to Parallel Simulation

Parsim is a hybrid of the time-driven and event-driven techniques [5]. In Parsim the system is modeled as a set of components interconnected via nets. Events, which are changes in the values of nets, activate actions associated with the corresponding nets, and the evaluation of these actions constitutes the simulation of the components. The kernel maintains a parallel event queue that is a time-ordered list of event lists. Each of the sublists contains all the events scheduled for the same simulation time. Special action lists are associ-
ated with the edges of the clock and contain actions to be performed when the value of the clock changes.

Parsim simulates events in time steps. At each step, it dequeues the first list of events from the system event list and evaluates these events (i.e., changes the values of the associated nets) concurrently. While Parsim evaluates the events, it constructs the list of actions activated by these evaluations. When all events have been processed, Parsim invokes in parallel all the actions in the constructed action list. If at the same simulated time a clock net value has changed, all actions in that clock net's action list are invoked. Evaluation of an action routine may result in new events posted on the global event list. The simulation terminates when there are not any events in the global event list or any actions on the clock action lists.

3 Implementation

3.1 The CARL Language

The CARL semi-abstract language is a C-like language in which the user can define components as functional blocks that respond to input stimuli by calculating new values and transmitting them to each other (behavioral components) [4]. CARL also assists hierarchical design by providing constructs for building components in terms of other components (hierarchical components). These two features of CARL allow reconfiguration of existing components into new simulations, as well as interchangeable use of multiple versions of the same component in order to explore architectural variations and designs. Moreover, using the CARL language reduces the effects of the implementation differences on the performance of the various simulation methods and allows the designer of simulation algorithms to study their inherent differences [1].

A component is described in CARL by using C-like code headed by CARL constructs keywords. The user has to define the inputs and outputs of each component (even though components may have no inputs or no outputs), its runtime initialization routine (if the user wishes to initialize the component prior to execution), and its creation routine (which creates an instance of the component and may perform one-time initializations). If the component is behavioral, the user defines a state for the component\(^1\) and the actions to be invoked when inputs change. On the other hand, if the component is hierarchical, the user defines its subcomponents, as well as how these subcomponents are connected to each other and to the inputs and outputs of the component. An example of a component definition is shown in Figure 1.

The definition of a component can be translated into C code, compiled into object code, and linked with the Parsim runtime library to form a Parsim simulator [5]. The same definition can also be translated into C++ code, compiled into object code, and linked with the PDES runtime libraries to form a PDES simulator. When the user executes the PDES simulator he can choose to use either the Chandy-Misra or the Time Warp approach. The user can make this decision after the simulator has been constructed but before it starts executing. In this way the user can experiment with the various techniques and use whichever is best suited to the particular simulation.

3.2 The Testbed

For our simulations we used an Alliant FX-80 [2], a shared memory machine with 8 processors. Each processor executes a single Concentrix process, a scheduler that switches between the various logical processes. In the Chandy-Misra variations we implemented, when a process is blocked, it returns control to the scheduler, which then selects a new ready logical process and transfers control to it. In the Time Warp approach, a logical process can also voluntarily return control to the scheduler which, in turn, inserts it into the ready scheduling queue and then transfers control to another ready logical process. We added this feature to the Time Warp model in order to implement a moving time window necessary in the simulation of synchronous systems, since the input of clocked components never becomes empty (the clock continuously generates new inputs).

We implemented our system using threads [3, 14, 22] because Concentrix processes are heavy and because using the operating system for scheduling would result in unacceptably high overheads. In our system, each logical process is a light-weight thread. Since no thread package is available for the Alliant, we implemented one toward the efficient execution of parallel discrete event simulations. Using the thread approach allowed us to experiment with specific runtime kernel operations such as scheduling without having to modify other parts of the kernel.

4 Partitioning

In the shared memory implementation of parallel discrete event simulators, message-based communication between the components is implemented via accesses

---

\(^1\)The state of a component is the data structure that represents the component in the simulation.
COMPTYPE q4i4o (ndId)
int ndId;
INPUTS
double in[INPUTS]: NewInput;
OUTPUTS
double out[OUTPUTS];
VAR
    int ID, numJobs;
    double oldIn[INPUTS];
    long nxtServiceT;
    double serviceTime;
    double waitTime;

numJobs = 0; nxtServiceT = 0;
serviceTime = 0; waitTime = 0;
for (i=0; i<INPUTS; i++)
    { in[i] = -1; oldIn[i] = -1; }
for (i=0; i<OUTPUTS; i++)
    out[i] = 0 after 0;
BEGIN
    ID = ndId;
END

ACTION NewInput
    int i; double srvcT; long delay;
for (i=0; i<INPUTS; i++)
    if (in[i] != oldIn[i]) { /* a new job arrived */
        numJobs++;
        oldIn[i] = in[i];
        /* service this job */
        srvcT = dRnDGen();
        serviceTime += srvcT;
        delay = (long)(srvcT*100);
        if (nxtServiceT <= now)
            { nxtServiceT = delay + now; }
        else {
            long tmpdelay = delay;
            waitTime += nxtServiceT - now;
            delay += nxtServiceT - now;
            nxtServiceT += tmpdelay;
        }
    /* send the job to an output port */
    out[iRnDGen()] = (in[i] + srvcT) after delay;
}

Figure 1: A Node of the Toroid Network in CARL

to shared message queues residing in the global memory. Since the message queues are shared between the producers and the consumers of the messages, accesses to the queue must be synchronized via a locking mechanism. But if we could determine that the producer and the consumer of a message could never be active at the same time, the synchronization would no longer be necessary. Therefore, clustering communicating components together and assigning each cluster of components to a processor reduces the synchronization overhead, since only intercluster communication requires synchronization [21]. Unfortunately, optimal partitioning is a hard (NP-Complete) problem, and only heuristics can be practical in deriving good partitionings.

4.1 Random Partitioning

Random assignment of components to processors is the easiest and fastest partitioning method. Components are chosen randomly and grouped together into a cluster. We assign about the same number of components to each of the clusters. Since we do not know the load of each component, we assume that all the components represent the same load. Consequently, having clusters with the same number of components enables us to balance the load across the processors. This method, however, does not consider the intercluster synchronization overhead, and thus frequently communicating components may be assigned to different clusters. Hence, random partitioning may result in high communication overhead during the execution of the simulation, which in turn results in longer execution times.

4.2 Partitioning Using Breadth First Traversal of the Graph

One way to achieve better clustering is first to combine communicating components into “super-nodes” in the simulation graph and then partition the resulting graph. We chose to combine components that form trivial cycles, i.e., directed cycles that include only two components, because they are tightly coupled and their concurrent execution would result in contention on the message queues synchronization locks. We did not combine more than two components that form a cycle, because in order to construct sets with the same cardinality we might have to split the “super-components,” and that would complicate the subsequent partitioning phase. We also decided not to combine frequently communicating components. If we assume that all communication channels are equally utilized, components that
are connected through many links are expected to communicate a lot more than the rest of the components. Combining these components would decrease the communication overhead. However, it could also limit the available parallelism, since the simulation of these components could not overlap in a pipelined form.

In order to exploit some kind of pipelined execution and, at the same time, reduce the communication overhead, we use a modified breadth first traversal to perform the partitioning of the graph. We start with an initial set of (possibly composite) components which we insert into a FIFO queue and consider the first partition. We repeatedly assign the first component on the queue to the partition we consider, enqueue all its unassigned successors into the queue, and consider the next partition. If we fail to assign a component to a cluster, we try to assign it to one of the other clusters. If we cannot assign it to any of the other partitions, we insert it into the first partition we considered for it, ignoring the limit on the maximum number of components in a cluster. By the time the queue becomes empty, all the components have been assigned to some partition and all partitions have about the same number of components.

4.3 Partitioning Using Depth First Traversal of the Graph

Obviously, the above method treats the graph as a "pseudo-layered" one and attempts to distribute the components on each layer to the available processors. It does not, however, take into account the intercluster communication overhead, except during the combining step. Another approach that considers the communication overhead is using a depth first traversal of the graph instead of the breadth first traversal. We start with the same initial set of components which we push onto a stack and consider the first partition. We repeatedly assign the top of the stack to the partition we consider, enqueue all its unassigned predecessors onto the stack, until we fail to assign a component to a partition. In this case, we consider the next partition and repeat the previous procedure. If we cannot assign the component to any of the partitions, we insert it into the first partition we considered for it, and consider the next partition for the remaining unassigned components. By the time the stack becomes empty, all the components have been assigned to some partition.

This method constructs chains of communicating components and assigns these chains to the available processors in an attempt to minimize the intercluster communication cost. However, the resulting partitioning will prove inefficient when the computation is performed along one or few of these chains, since the load will be unbalanced across the processors.

4.4 Manual Partitioning

In several situations one of the above procedures performs better than the others, but there are also situations where none of them performs satisfactorily. Manual partitioning seems to be the best solution to the partitioning problem. After all, the user knows the topology of the simulation graph and, probably, how the computation will develop. Thus, he or she can derive a partitioning that minimizes intercluster communication overhead and, at the same time, does not eliminate potential parallelism. Indeed, manual partitioning is the best approach for small or medium size simulations on a small number of processors, where the user can actually derive the partitions. However, in the case of big simulations with few hundreds or thousands of components, or when the structure of the graph is not regular, manual partitioning is a very difficult and painful task, because there are many conflicting factors the user has to balance for each connection. In this case, manual partitioning is most likely to produce poor results, so it is inappropriate for big (real) simulations.

5 Dynamic Scheduling for the Chandy-Misra Method

The efficient implementation of the PDES methods on distributed memory machines depends on the existence of a good static partitioning technique. The reason is that migrating data and processes from one processor to the other is an expensive operation and cannot be used frequently. Shared memory machines, however, allow processor self-scheduling to perform equally well or, in some cases, even better. When all the components in the simulation represent almost the same load and are equally utilized, dynamic scheduling does not perform as well as static partitioning due to the synchronization overhead required to ensure mutual exclusive access to the shared message queues. On the other hand, when the components do not represent the same load or are not equally utilized, dynamic schedul-
ing performs better than static partitioning, because it balances the load across the participating processors. In our implementation we decided not to use the operating system or a global scheduler to perform the scheduling because the associated cost would be prohibitively high. Instead, we used processor self-scheduling, which has lower overhead and can achieve better load balancing [16].

The first approach we implemented maintains a separate scheduling queue for each processor [3]. When a simulated component is unblocked, it is inserted into the scheduling queue of the processor that unblocks it. When a processor becomes idle, it checks its own scheduling queue to find a component to simulate. If its queue is not empty, it dequeues the first component and simulates it. Otherwise, it checks the scheduling queues of the other processors for work. If a ready component is found, the processor grabs it and simulates it. If there is not any ready component on any of the scheduling queues, the processor repeatedly idles for a small period of time and reexamines the scheduling queues of the other processors until it finds a ready component.

The second approach we implemented also requires a scheduling queue per processor. A processor acquires new work in the same way as in the previous method. In this approach, however, each processor inserts unblocked components to all the scheduling queues visiting one ready queue after the other in a circular fashion. In this way we avoid the problem of one or few of the queues becoming the bottleneck of the system when only one or few of the processors unblock components, but we increase the probability of contention on the synchronization locks of the scheduling queues. This approach is a modified version of the cooperative scheduling studied in [18].

The third approach we implemented attempts to decrease the contention on the synchronization locks of the scheduling queues. Each processor maintains two scheduling queues: a private queue which can be accessed only by itself, and a shared one which can be accessed by any processor. Some of the components a processor unblocks are inserted on its private queue and the rest of them are enqued on its shared queue. Whenever a processor becomes idle, it checks its private queue for work. If it finds a ready component, it dequeues it and simulates it. Otherwise, it examines the shared scheduling queues, starting with its own. If there is not any ready component on any of the scheduling queues, it repeatedly idles for a while and reexamines the shared queues of the other processors until it finds a component to simulate. The performance of this approach depends on whether a processor finds a component on its private queue whenever it checks for work, or it has to search the shared queues for ready components. This, in turn, depends on how often a processor inserts an unblocked component on its private queue. There is a trade-off between synchronization overhead and load imbalance in the choice of how often a processor should use its private queue.

6 Time Warp Scheduling

Time Warp scheduling is more complicated than scheduling in a Chandy-Misra simulator, because selecting the erroneous computations to proceed while correct ones wait on the scheduling queue affects the performance of the system. However, it is not possible to know beforehand which computation is correct and which is erroneous. We can tell, though, that a computation scheduled for an earlier simulation time is more likely to be correct than one scheduled for a later simulation time. Therefore, by discriminating against computations that are more advanced in simulated time, we increase the possibility that correct computations will proceed faster than the erroneous ones, and that the simulation will terminate in less (wall clock) time.

One way to achieve this discrimination is to use a priority scheduling queue, where the priority of each ready component is the time stamp of the first message on its input message queue, and schedule next the component with the smallest priority. Using a simple priority queue, however, may result in large scheduling overhead, since we have to synchronize the accesses to the shared data structure. For this reason, we used a parallel priority queue, where multiple insertions and deletions can proceed concurrently, lowering in this way the scheduling overhead [19, 20]. Even though priorities are only approximations of where the LP actually is in simulated time, for most of the time this scheduling strategy selects for execution LPs that are furthest behind in the simulation. This selection helps the Time Warp simulator to execute faster because it executes fewer erroneous computations and overhead operations.

7 The Benchmark Programs

In order to study the performance of the implemented simulation methods, we examined two kinds of systems. First we simulated a synchronous system: a 16-processor/16-memory system with an Omega interconnection network [15]. Each processor calculates requests for the memories and transmits them to the appropriate memories through the forward Omega net-
work. After each memory performs the requested operation, calculates the delay of the request through the network, and updates its statistics, it sends the result of the operation to the requesting processor through the backward Omega network. Upon receiving the answer, the processor calculates the delay of the answer through the backward network and updates its statistics. A processor can generate random, linear barrier synchronization, or vector requests, each request being a read from or write to the memory. The Omega networks are packet switched and each packet consists of two to four 32-bit words. The $2 \times 2$ switches of the forward network are implemented using two input registers and two output buffers, whereas the $2 \times 2$ switches of the backward network have an additional table to support broadcasting. Because this simulator has been written for the Parsim simulation method, the processors, memories, registers, buffers, and tables are designed as finite state machines. In the PDES simulator, each of them is represented by a separate logical process.

We simulated the multiprocessor system using three traffic models: random traffic, in which each processor generates requests as fast as possible to randomly chosen memories, hot spot traffic, in which each all the processors perform operations on the same memory module, and conflict-free traffic, in which each processor issues requests to the memory module at the same address as the processor. We also performed an experiment in which a processor idles for a while when it generates a memory request (non-uniform load). In all the cases, the method described in section 3.2 (BFT) achieves the best performance among all the static partitioning methods (Figure 2). Random partitioning always results in the worst performance. Sometimes manual partitioning performs well, but when the components cannot be mapped nicely to the processors, it does not perform so well. Finally, the method described in section 3.3 (DFT), performs well in all cases, but always worse than the BFT method; because of the large number of components per partition, there is very little contention on the synchronization primitives, and therefore there is no advantage in using this method.

On the other hand, all dynamic scheduling techniques perform equally well in all the cases (Figure 3). Due to the large number of components per partition, each processor almost always finds a ready component on its scheduling queue. Therefore contention on the scheduling queues, which would have otherwise resulted in different performance between the various methods, does not occur.

In the first three experiments, Parsim outperforms both the Chandy-Misra and the Time Warp methods (Figure 4). In order to avoid the deadlock, the Chandy-Misra methods have to advance the time for all the components, even when only a few of them are simulated. The large number of components, as well as the small CPU time used to simulate an event, make the overhead noticeable and the methods impractical for these kinds of simulations. Time Warp, on the other hand, performs slightly better than Chandy-Misra in the cases of random and conflict-free traffic models, and much better in the hot spot traffic case. The main reason that Time Warp does not perform as well as Parsim is the state-saving overhead. Each of the components in this simulation has many state variables, so that when many components are simulated, as in the cases of the random and the conflict-free traffic models, the performance of Time Warp is not so good, whereas when few components are simulated, as in the case of the hot-spot traffic, Time Warp performs better and approaches the performance of Parsim.

In the fourth experiment, the relative performance of the different methods varies. Parsim does not perform so well, because it has to simulate all the processor components at the same time step and consequently the processors that finish this step earlier have to wait for the rest of the processors to finish simulating components at the current step before continuing to the next one. The Chandy-Misra method performs very well when we utilize dynamic scheduling. Simulation of the more loaded components is overlapped with the advance of time for the components that are not simulated, and the computation is well balanced across the processors. However, when we use static partitioning, the Chandy-Misra method does not perform so well in all the cases, and its performance depends on how well the load can be distributed over the processors (see the 7-processor case in Figure 4). Finally, Time Warp performs better than all the other methods. The state-saving overhead is hidden by the time taken to simulate the more loaded components, and its performance scales well with the number of processors.

The second system we simulated was a $4 \times 4$ toroid network, a benchmark widely used in the performance studies of PDES methods [9, 23]. Each node for each incoming job calculates the earliest available service time and a random service period. Then it chooses an output port and sends the job to a neighboring node. The total message population remains constant at one

\[140\]
Figure 2: Omega Network, Static Partitioning
Figure 3: Omega Network, Dynamic Scheduling
Figure 4: Omega Network, Chandy-Misra, Time Warp, Parsim
message per link throughout the simulation.

We simulated the system using four different distributions: biased, exponential, uniform, and deterministic. The speed up achieved is calculated as the execution time of the Parsim simulator on a single processor divided by the execution time of the parallel simulator. Even though Parsim does not have the same implementation as the Chandy-Misra and Time Warp parallel simulators, its execution on a single processor can be considered as a conventional sequential event list simulation. Still, the results are optimistic, because sequential simulators with event lists implemented as splay trees are faster than the simulators using simple linked lists. In some cases (e.g., Figure 6), the speed ups are superlinear due to the way Parsim executes the simulation.

The speed up curves for the different distributions are shown in Figure 5. The performance of the Chandy-Misra simulator using dynamic scheduling in the biased and deterministic cases is very good, but not so good in the exponential and uniform cases. In the deterministic case, the speed up obtained is not as good as the biased case because Parsim performs better when all events are on the same time step than when the events are scattered on different steps. In the exponential and uniform cases, the speed ups obtained are smaller because the lookahead of each process is much smaller. The Chandy-Misra simulator using static partitioning has very poor performance in all the cases because each processor is constrained to execute very few components and, therefore, most of the time is spent on the synchronization primitives of the message queues. Both the aggressive and the lazy cancellation Time Warp simulators perform well on all distributions. The state of each component is small, and therefore the state-saving overhead is small compared to the amount of computation performed. Parsim performs very well in the deterministic case because all the events occur at the same time step, and consequently parallelism is available to exploit. However, its performance in the other cases is very poor because there is almost always a single event on each time step, and Parsim can only exploit the parallelism available on each time step.

Next, we performed the same experiments using larger grain of computation per simulated event. The speed ups obtained (Figure 6) are essentially the same as in the first set of experiments, except that in this case the Chandy-Misra method using static partitioning performs a little better because the computation over communication ratio is larger. Finally, we performed the same experiments with few nodes having larger grain of computation per message than the rest of the nodes. The results of these experiments (Figure 7) are in essence the same as the results in the second set of experiments and do not provide further insight into the performance of the various simulation methods.

8 Conclusions

This paper describes the implementation of the Chandy-Misra and the Time Warp parallel simulation methods on a shared memory multiprocessor. It also examines the performance of these methods in the simulation of both synchronous and asynchronous systems and compares their performance to a hybrid simulation method.

The results obtained show that careful static partitioning is important in the Chandy-Misra method, and that dynamic scheduling should be used when the load of the components in the simulation varies. Even though good speed ups are obtained for the asynchronous system, the performance of both methods is not as good in the simulation of synchronous systems. The Chandy-Misra method seems to be totally inappropriate for the simulation of synchronous systems, but the Time Warp approach achieved good results when the state-saving overhead to the computation-performed ratio was reduced. On the other hand, Parsim performs well in the simulation of synchronous systems, since it has been specifically designed for such systems, but does not perform well in the simulation of asynchronous systems, where there is not much parallelism available at each time step for it to exploit.

We plan to continue our studies on the performance of the PDES methods in the simulation of synchronous systems. Although such systems contain significant parallelism, the PDES methods are not able to exploit it. We will also continue to experiment on shared memory machines because they provide the means for an efficient implementation of the PDES methods.

References

Figure 5: Toroid Network, No Extra Load for Each Node
Uniform Distribution

Deterministic Distribution

Biased Distribution

Exponential Distribution

Figure 6: Toroid Network, Uniform Load for Each Node
Uniform Distribution

Deterministic Distribution

Biased Distribution

Exponential Distribution

Figure 7: Toroid Network, Nonuniform Load


