Parallel Simulations of Multiprocessors *

Pavlos Konas and Pen-Chung Yew
Center for Supercomputing Research and Development,
447 CSRL, 1308 West Main Street,
Urbana, IL 61801.
konas@csrd.uiuc.edu
phone: 1-217-244-0046

1 Introduction

In recent years, advances in technology and in the area of computer architecture have resulted in increasingly more complex designs. As the complexity of the designs increases, it becomes more and more difficult to study their behavior using analytically tractable performance models. Simulation emerged as the only viable method for gaining insight into the operation of new as well as existing computer systems. However, detailed simulations of computer systems require excessive amounts of memory and 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.

Most of the existing parallel methods were developed for simulating asynchronous systems [13]. In such systems only a few evaluations occur at any single point in simulated time; a parallel simulation method has to find and expose the parallelism for efficient parallel execution. In contrast, architectural simulations are inherently synchronous; the activity during the simulation follows the changes of the clock signal(s) (a behavior known as the clock effect [32, 39, 40]). Furthermore, such simulations contain significant amounts of inherent parallelism [36]. Hence, the main problem in architectural parallel

*This work was supported in part by the National Science Foundation under Grant Nos. MIP 89-20891 and MIP 93-07910, NASA NCC 2-559, and a donation from Motorola.

simulations is to exploit the available parallelism with the least possible overhead. Unfortunately, both conservative and optimistic asynchronous methods introduce significant overhead in their attempt to find parallelism [18, 19, 36]. They are not appropriate for synchronous architectural simulations. Therefore, we focus our attention on synchronous parallel simulation methods.

In this paper we present a synchronous, parallel, event-driven approach (SPaDES) to the execution of a single simulation program on a parallel machine and study its performance. The method models the simulated system as a set of communicating logical processes (LPs) and exploits the parallelism available at each point in simulated time. It differs, however, from previous approaches in many ways: (1) it contains a single, global synchronization operation per iteration; (2) it introduces a minimum number of kernel operations into the simulation; (3) it allows for efficient processor self-scheduling; and (4) it aggressively exposes parallelism only when there is not enough parallelism inherent in the simulation for the processors to exploit.

In our study, we simulate a *synchronous* multiprocessor machine. The most important characteristic of such systems is that the majority of the events occur during small time intervals that contain the clock edges, while for the rest of the time there is only a limited number of events to simulate. Other characteristics of such systems include the large number of components of possibly variable load, and the large state size of those components.

We study the performance of the simulation method on both a centralized shared-memory multiprocessor (an Alliant FX/80) and on a simulated NUMA (Nonuniform-Memory-Access) multiprocessor [6]. The centralized memory of the Alliant reduces the cost of synchronizing the processors, simplifies the load balancing during each time step, and decreases the cost of communication between the LPs. On the other hand, the NUMA characteristics of the simulated multiprocessor give us an opportunity to study the performance of the method under more demanding run-time conditions and to assess the impact of additional issues, such as partitioning and processor self-scheduling, on the efficient execution of a parallel simulation. In all the experiments, SPaDES is compared against a traditional synchronous parallel simulation method. The few kernel operations introduced by SPaDES, the use of a single barrier during each time step, and the exploitation of a coarser-grain parallelism are the main reasons SPaDES achieves its high performance.

The object oriented design of the simulator and the logical processes model on which it is based allow us to easily port the simulator on a wide range of multiprocessors: we only need to express the primitives for parallelism and synchronization in the simulator using the primitives provided by each multiprocessor. Of course, for maximal efficiency different methods for partitioning and scheduling should be utilized, but such methods are already available within the simulator framework.

Our main concern in this paper is the performance of the proposed parallel method in the simulation of synchronous architectural designs. Section 2 gives a brief overview of related work in both synchronous parallel simulation methods and in parallel architectural simulations. In Section 3 we present the synchronous PDES method. In Section 4 we describe the simulated system and in Section 5 we present the experiments performed on the Alliant and NUMA multiprocessors and the results we obtained. Finally, Section 6 contains our conclusions.

2 Related Work

2.1 Synchronous Parallel Simulation Methods

In synchronous parallel simulation methods, one iteratively determines which events are safe to process and then processes them [13]. Barrier synchronizations are used to keep iterations (or components of a single iteration) from interfering with each other. The feature that separates different synchronous approaches is the method used to determine which events are safe to process.

Najjar et al. proposed the use of *directional distances* between objects to identify independent events on a centralized event queue [29, 28]. Two events are independent if the interval of time between their respective occurrences is smaller than the minimum information propagation delay between the two objects to which they belong. Each iteration of the algorithm is divided into three phases. The algorithm first identifies a set of consecutive and pairwise independent events starting with the head of the event queue; it then schedules the concurrent evaluation of these events on separate processors; and finally, new events generated during the concurrent evaluation phase are inserted into the event queue.

Ayani proposed a similar algorithm, called the *three phase algorithm* [2, 3]. In this algorithm, however, each object maintains its own local event queue and is responsible for deciding whether or not to evaluate an event from its input. A component bases its decision on the timestamp of the event, its directional distances to the other objects in the simulation, and their local virtual times. Each iteration of the algorithm, first identifies the *definite* independent events from a set of potentially independent events, and then evaluates these events concurrently.

Lubachevsky proposed the use of a moving time window to reduce the overhead associated with determining when it is safe to process an event [23, 24, 25]. The algorithm, called the *bounded lag algorithm*, determines which events occur within a statically defined window and executes them concurrently. The algorithm then recomputes the lower edge of the window as the minimum timestamp of any unprocessed event (thus advancing the window in simulated time) before repeating the above steps. The simulation terminates when the lower edge of the window reaches the end of simulated time.

Nicol proposed an algorithm similar to Lubachevsky's bounded lag, but in which the width of the execution window is dynamically calculated [30]. At the beginning of each iteration, the processors cooperatively determine the upper window edge (*ceiling*). This value is chosen in such a way that all events within the time interval $[floor, ceiling)$ can safely be processed in parallel. After the processing of the events has completed, *ceiling* becomes *floor* for the next window and the iteration is repeated.

Other interesting synchronous methods proposed in the literature are *Space-Time* [11], the use of *conditional knowledge* [12], both of which were proposed by Chandy and Sherman, and *Breathing Time Buckets* a hybrid method proposed by Steinman [37, 38]. However, the most obvious and commonly used parallel synchronous simulation algorithm is the straightforward parallel implementation of the traditional event driven algorithm [9, 35, 36]. Under this algorithm, simulation evolves in steps which are divided into two phases. During the first phase, we concurrently evaluate all events occurring at the earliest simulation time. Then, during the second phase, we simulate in parallel all the components whose inputs have changed by an event evaluation. Barrier synchronizations between the two phases guarantee the correctness of the simulation. An efficient implementation of this parallel centralized-time algorithm is Parsim, a simulation method developed at CSRD [8, 9]. Because of its efficiency, we use Parsim as the basis of our study of the proposed parallel simulation method.

2.2 Parallel Architectural Simulations

As computer systems become increasingly more complex, studying their architectures and exploring design alternatives through simulation becomes increasingly more computation demanding. Such systems are naturally discrete and inherently parallel. Thus, they are excellent candidates for parallel simulation. Several parallel simulation methods have been used with various degrees of success.

Yu et al. [41] use a time-driven algorithm to simulate a class of Multistage Interconnection Networks (MINs) called *Buffered Delta Networks*. In their work, they focus on the synchronization of the processors within and between time steps and present three approaches for handling the problem: multibarriers, locks, and temporary storage. The multibarrier approach achieves the best performance among the three alternatives and delivers speedups between 4 (for a 4-stage network), and 7 (for a 9-stage network), on an 8-processor Sequent Symmetry. In a follow-up paper [15], they present an improved processor allocation and synchronization algorithm, called the *two-phase algorithm*, which exploits the network's topological structure. The speedups achieved by this algorithm on a 16-processor Sequent Symmetry range between 8 (for a 5-stage network), and 14 (for a 9-stage network).

Ayani and Rajaei [4] use the three-phase algorithm to simulate a class of MINs called *Generalized Cube Networks* (GCMINs) on a 16-processor Sequent Symmetry. The speedup they obtain increases as the size of the MIN increases and becomes almost linear for MINs with more than 256 inputs. The relative speedups they present are lower than the absolute ones and range between 1 and 9 for the largest simulated MIN (512-input). In a follow-up paper, Ayani and Berkman [5] simulate GCMINs using the three-phase algorithm on a CM-2 with 8,192 processors. Even though they achieve speedups of more than 2,000 for networks with 14 stages, the efficiency of their implementation is considerably low.

Konas and Yew [19] compare the performance of a parallel centralized-time event driven method to the Chandy-Misra [10] and Time Warp [17] methods in simulating a multiprocessor. The results they present show that the synchronous method is considerably faster than both the Chandy-Misra and the Time Warp approaches. The clock advancing overhead introduced by the deadlock avoidance mechanism in the Chandy-Misra method makes this approach inappropriate for architectural simulations. Time Warp, on the other hand, introduces the state saving overhead and is not expected to perform well in detailed architectural simulations because such simulators tend to have components with large state sizes. However, in an associated report [18] Konas observes that the efficiency of the synchronous method is considerably low. He attributes this problem mainly to the scheduling policy the method employs and partially to the underlying synchronous model.

3 An Improved Synchronous Parallel Simulation Method

All previous synchronous parallel simulation methods attempt to reduce the overhead associated with finding parallelism, while remaining aggressive in exposing it. However, they introduce global synchronization operations in order to coordinate the processors executing the simulation. Global synchronizations have several implications on the execution of a parallel program. They introduce run time overhead which can be significant if the operation is not supported in hardware. They make load balancing more critical for the performance of the program and more difficult to achieve. They make the algorithm more sensitive to practical considerations such as page faults, cache effects and operating system interrupts. And most importantly, scaling the algorithm to a larger number of processors becomes harder.

Furthermore, all methods use some mechanism to determine which events are safe to process concurrently during each iteration. The more aggressive a method is, the more sophisticated the mechanism it uses to expose parallelism, and the more overhead it introduces into the computation. This overhead usually depends on the number of components in the simulation. In some methods, however, it also depends on the structure of the simulation graph. For architectural simulations, in which the number of components is large, one would like to avoid as much as possible such overheads because they reduce significantly the effectiveness of the parallel simulation.

Among all previous synchronous methods, the centralized-time event-driven algorithm seems to be the most appropriate method for parallel architectural simulations. However, its implementation contains potential overheads which may reduce its efficiency. First, each iteration of the algorithm contains two barrier synchronizations: one after the event evaluation phase and the other after the component simulation phase. Second, the event evaluation phase exploits very fine grain parallelism which does not match well the computation model supported by most multiprocessors. Third, the overhead introduced by the method in the event evaluation phase exceeds the processing requirements of that phase. Fourth, the simulation potentially exhibits poor locality of references. And finally, the use of two conceptually global data structures (the event queue and the activation list) lengthens the serial parts of the computation and introduces potential contention in the parallel portions of the execution.

In order to avoid these potential problems and reduce the overhead introduced by the simulation method, we propose a conservative synchronous parallel simulation method (SPaDES) [20]. Our approach is based on the logical processes model [13]. The simulator consists of a set of communicating logical processes (LPs), each of which represents a component or a group of components of the simulated system. Each LP has a single, time-ordered, input queue which acts as its local event list. Each event represents any and all changes in the value of the input variables of the LP occurring at the same simulated time. The creation of an event *activates* the LP for the simulation time of the event's occurrence and results in the insertion of the LP into the scheduling queue of the processor that activated it. Each *processor* maintains its own scheduling queue, which is an ordered list of LP lists. Each of these sublists contains the LPs scheduled for a particular simulation time. Figure 1 shows the execution of a SPaDES simulation on two processors. As we can see, *LP1* has three events in its input queue: $< 4, \_ >$, $< 5, a >$ and $< 2, z >$ which are to occur at the times 2, 4 and 6 respectively. Its current input state is $< 1, k >$. The scheduling queues of the two processors simulating the system are shown in the lower part of the Figure.

Each input of a component may have an action associated with it. An action is a function that describes the behavior of the LP in response to changes in the value of an input. During its simulation, an LP changes its input state by evaluating the event on the head of its event list. If an input associated with an action changes during this evaluation, the LP invokes the corresponding action, thus simulating itself for the current simulation time.

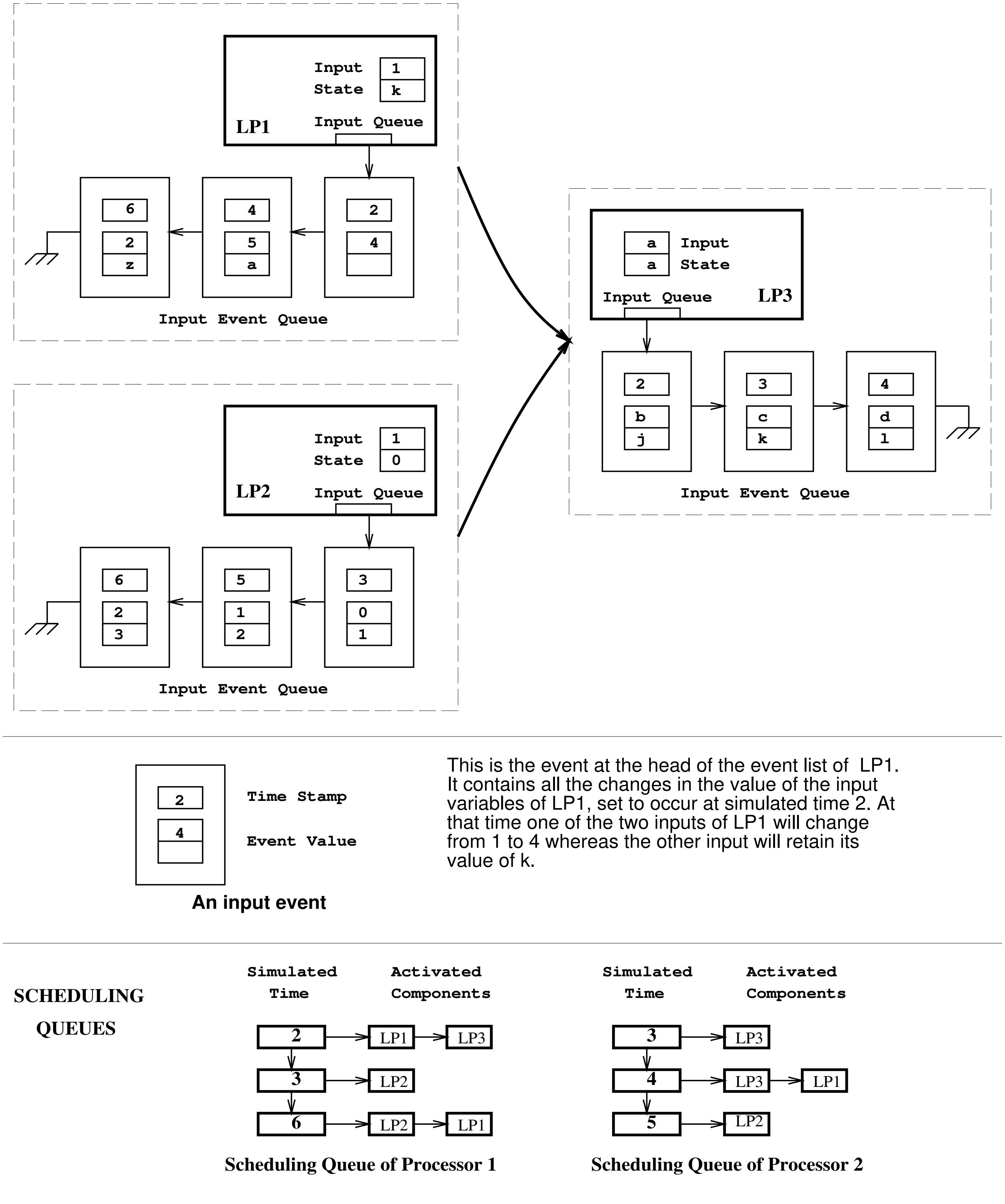


Figure 1: A Snapshot of a SPaDES Simulation

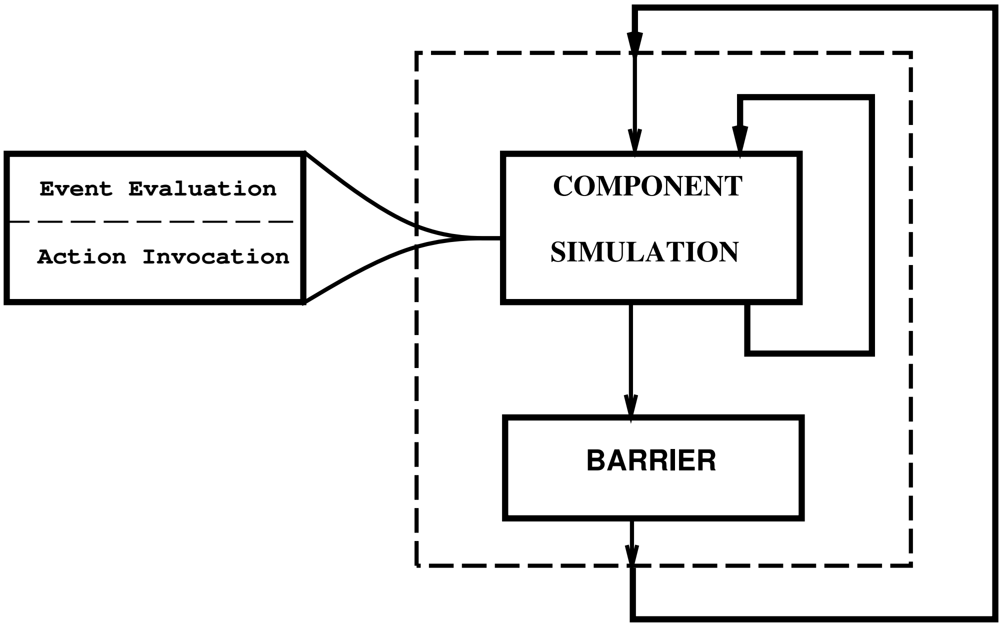


Figure 2: The SPaDES Execution Model

The simulation evolves in steps (Figure 2). At the end of each step, the processors synchronize with each other by means of a barrier. As each processor reaches the barrier, it participates in deciding the simulation time of the next step. The last processor to arrive at the barrier sets the global simulation time of the next step. During a step, each processor simulates the components assigned to it which are scheduled for the current simulation time. When there are no more local active LPs to simulate a processor acquires and simulates remote LPs, thus balancing the load of the current iteration and minimizing its idle time at the barrier. When all LPs which are active during the current iteration have been simulated the current step terminates, and the processors reach again the synchronization barrier. The simulation terminates when there are no more active components on any of the scheduling queues, or when we reach a user-provided End_of_simulation_time.

An issue critical to the performance of a synchronous parallel simulation method is processor self-scheduling. The scheduling queue in SPaDES is similar in functionality to the event list employed in traditional event driven simulations, and to the scheduling queue of self-scheduled parallel programs. We use two different approaches to processor self-scheduling depending on the nature of the multiprocessor on which we execute the parallel simulation. On centralized memory multiprocessors we use per processor private scheduling queues and a single global queue shared by all the processors. The private queues allow us to avoid synchronization during the scheduling of LPs, whereas the shared global queue allows us to balance the load across the processors minimizing their idle times at the barrier. On NUMA multiprocessors, processor self-scheduling has the additional responsibility of addressing the tradeoff between load balancing and locality management [26, 27]. For this purpose we use a shared scheduling queue per processor and employ a two-level data structure that allows us to efficiently schedule conditionally active LPs for future execution and, at the same time, to easily exploit affinity information during the allocation of processors to unconditionally active LPs.

One important aspect of our approach is that we treat the clock input of an LP as an "internal" signal to the component, and we employ a lazy-type of evaluation in order to determine its value. This approach does not affect the correctness of the simulations since the clock inputs are needed only to synchronize the simulated components into a global time frame. However, it reduces significantly the number of events that need to be evaluated during the course of the simulation, thus increasing the efficiency of its parallel execution.

4 The Simulated System

In order to study the performance of SPaDES, we examine a behavioral model of a *synchronous* multiprocessor: a 16-processor/16-memory system with an Omega interconnection network [21]. This simulation models many parallel shared memory architectures found in practice [14, 16, 33, 31].

During the simulation each processor calculates requests for the memories and transmits them to the appropriate memory modules through the forward Omega network. After each memory module performs the requested operation, calculates the delay of the request through the forward 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 and updates several of its statistics which characterize the behavior of the network. A processor can generate random, linear barrier synchronization, or vector requests, each request being a read from or a write to the memory. The Omega networks are packet switched, and each packet consists of two to four 32-bit words. The 2×2 switches of the forward network are implemented using 2 input registers and 2 output buffers, whereas the 2×2 switches of the backward network include an additional table module to support broadcasting.

5 The Experiments

We simulated the multiprocessor system by employing three traffic models: *random* traffic, in which each processor generates requests to randomly chosen memories, *hot spot* traffic, in which 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 a fourth experiment in which each processor remained idle for a while after generating each random memory request. As a result of this idling period the processors represented heavier load than the rest of the components in the system (*non-uniform load*). In practice such a situation arises in trace driven simulations of multiprocessor systems where the processor modules read the traces from the disk, and in execution driven simulation of multiprocessors where the processors are actually executing an application program. In all four simulations, the processors generate requests as fast as possible (one request every four clock cycles), injecting, in this way, heavy traffic into the network.

We use Parsim as the basis of our evaluation of the SPaDES method. Parsim is a highly optimized parallel implementation of the centralized-time simulation method targeted for the efficient parallel execution of synchronous simulations. As such, it provides us with a good basis that strengthens the validity of our experimental results. Note that our approach is not the same as in other studies, where they compare a parallel method only to the sequential event-driven method.

5.1 Experiments on Alliant FX/80

Figure 3 shows the absolute speedups achieved by SPaDES and Parsim on an Alliant FX/80 in the experiments we performed. The absolute speedup, S_p, is calculated as the ratio of the execution time of our best **sequential** simulator to the execution time of the parallel simulator on p processors. In all the experiments the SPaDES method outperforms Parsim. SPaDES exhibits a better performance because it employs a single global synchronization operation during each time step, it introduces fewer kernel operations, and it exploits a coarse-grain parallelism (which suits well Alliant's computation model).

The speedups achieved in the random traffic simulations by both methods are similar to the corresponding speedups achieved in the conflict-free traffic simulations. The reason is that both traffic models generate similar activity in the system and therefore comparable amounts of parallelism during each time step. It is interesting to observe that in the conflict-free traffic simulations, Parsim achieves smaller speedups than in the random traffic simulations, even though the former simulations contain more parallelism during each time step than the latter ones. In addition, the speedups obtained by Parsim in the hot-spot simulations are comparable to those obtained in the previous two cases. The main reason for this outcome is the trade-off between the parallelism exploited and the overhead introduced by Parsim. The more events in a simulation, the more the parallelism exploited by Parsim, and the larger the speedup ob-

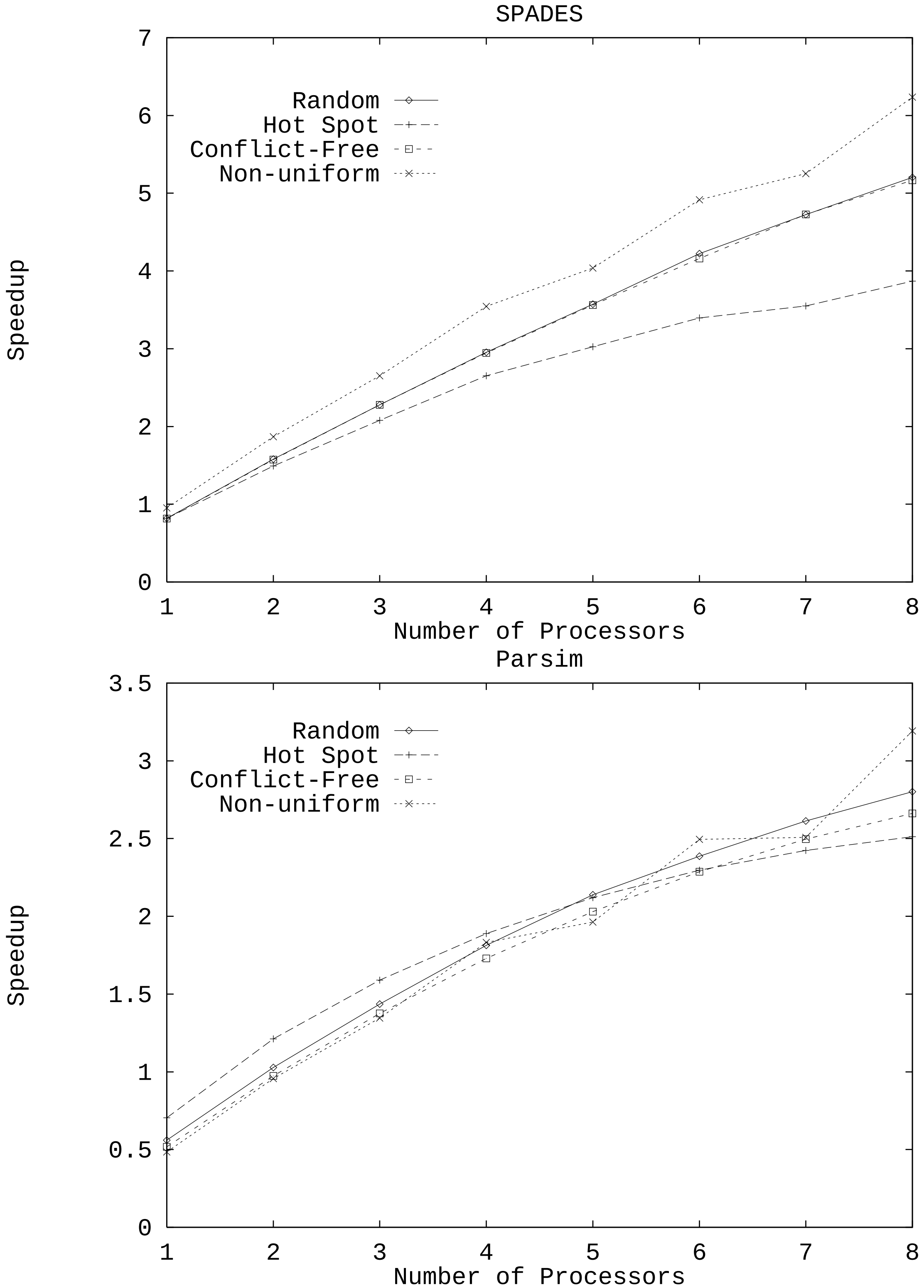


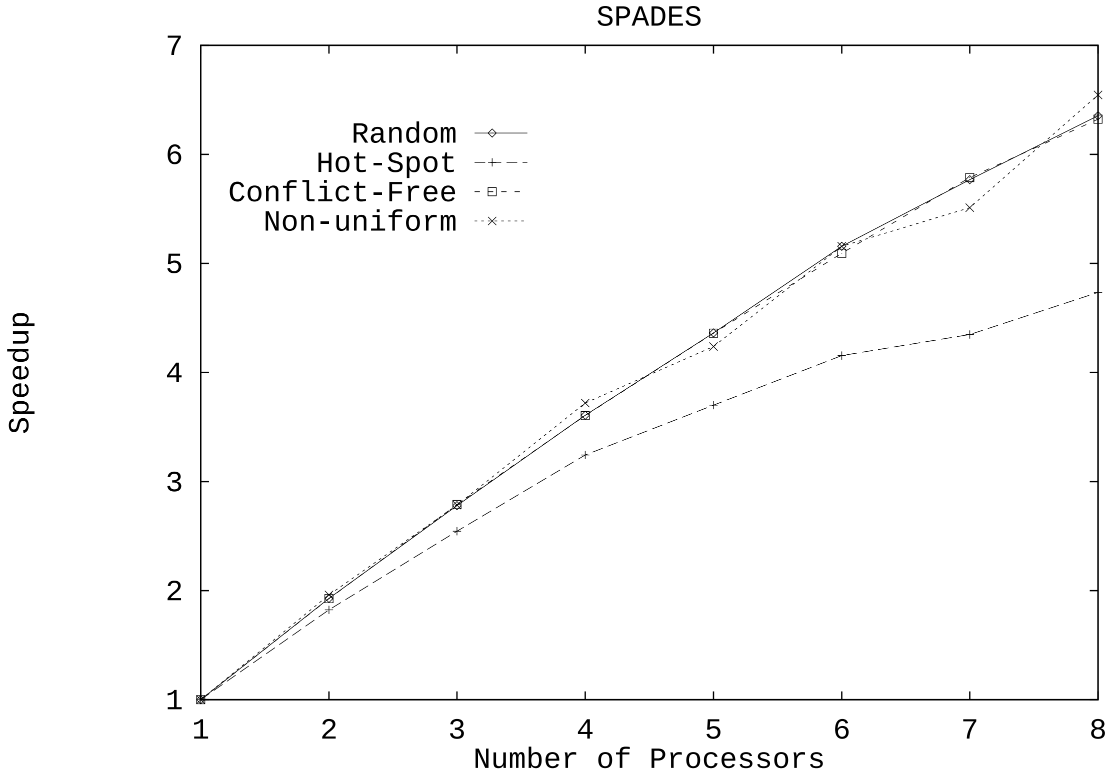
Figure 3: Speedup over the Best Sequential Simulator (Alliant FX/80)

tained over the sequential execution. On the other hand, the more events in the simulation, the larger the overhead of the Parsim simulator compared to the sequential one, and the larger the slow down compared to the sequential execution. Finally, in the non-uniform load simulations Parsim sometimes achieves better performance than in the other three simulations (6 and 8 processors cases) and sometimes its performance drops (5 and 7 processors cases). In this experiment most of the execution time is spent in simulating the processor components and, hence, the key issue is balancing the load of the component simulation phase across the processors. The speedup curve for this experiment is not as smooth as in the other experiments because Parsim is not always successful in balancing the work across the processors.

SPaDES, on the other hand, does well in the random and conflict-free simulations, where there is a moderate number of active components during each step. However, in the hot-spot simulation, where the number of active components during each step is small, SPaDES does not perform so well. The main reason is that the amount of exploited parallelism is not sufficient to hide the overheads introduced by SPaDES, such as the sequential portion of the computation and the barrier synchronization. Finally, the performance of SPaDES in the non-uniform load simulation is better than in the other three experiments, because the overhead SPaDES introduces constitutes a smaller portion of the total execution time than in the other simulations and because SPaDES balances the load well across the processors.

The importance of the number of kernel operations on the execution times achieved by the two simulation methods becomes evident in the hot-spot simulation. In this simulation the relative difference in the performance of the two simulators is smaller than in the other simulations. The reason is that the decreased activity in the studied system (compared to the activity in the other simulations) results in a larger decrease in the number of kernel operations executed by the Parsim simulator than the decrease in the number of kernel operations executed by the SPaDES simulator.

Figure 4 shows the relative speedups of the two methods in the various simulations we performed



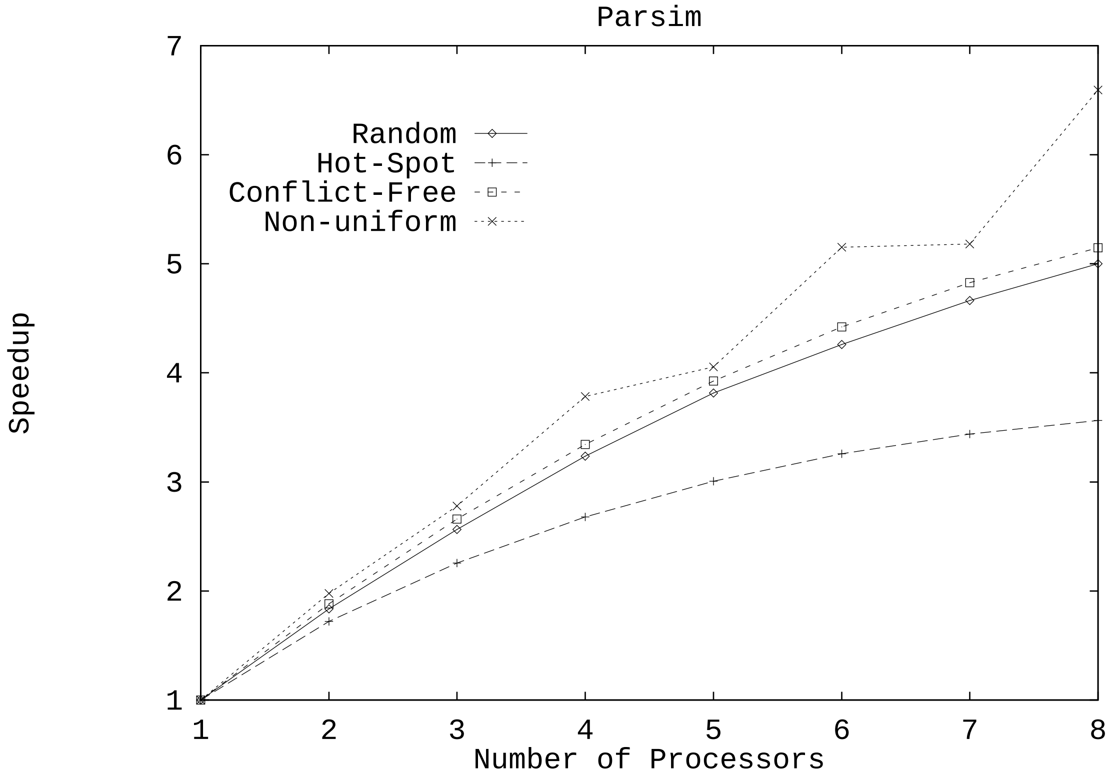


Figure 4: Speedup over the One Processor Execution (Alliant FX/80)

on the Alliant FX/80. The relative speedup, RS_p, is calculated as the ratio of the execution time of the parallel simulator executed on one processor to the execution time when p processors are employed. The relative speedups constitute a measure of the efficiency of a parallel simulator and impose an upper bound on the absolute speedups it can achieve.

The relative speedup curves of SPaDES are similar to its absolute ones because both the sequential and the parallel simulators share the same underlying model; only their slopes are steeper and the absolute values larger. On the other hand, the relation between the Parsim relative speedup curves is totally different from the relation between its corresponding absolute ones. The hot-spot simulation, which contains the lowest amount of parallelism, has the lowest efficiency, whereas the conflict-free traffic simulator, which contains the largest amount of parallelism, has the largest efficiency among the first three simulators. The main reason for this change is that the relative speedups factor out all the overheads associated with the kernel operations, and hence the more parallelism the simulated system contains, the better the speedup we obtain. Parsim achieves the best relative performance in the non-uniform load simulation because this is a simulation-intensive experiment which hides the overheads introduced by Parsim. However, Parsim's efficiency depends heavily on how well it distributes the work across the processors, and this is the reason its efficiency drops significantly in the 5-processors and 7-processors non-uniform load experiments. Finally, we should notice that the efficiency of Parsim in the first three simulations is still considerably lower than the efficiency of the SPaDES method, and as the speedup curves for the non-uniform load simulations show, it is harder to balance the load across the processors in Parsim than it is in SPaDES.

5.2 Experiments on a NUMA multiprocessor

Alliant FX-80 is a unique machine in that not only the memory is equidistant from all the processors but there is a single cache shared by all the processors. However, such a design is very difficult to scale to a large number of processors. Future machines will exhibit some form of NUMA (nonuniform-memory-access) characteristics (e.g. private caches, local memories, cluster memories etc.). The nonuniformity of the memory access delays makes an efficient implementation of a parallel synchronous method much harder. In order to examine the behavior and performance of our method on a NUMA machine, we implemented SPaDES on a simulated NUMA multiprocessor. We used Proteus [6] to simulate a multiprocessor consisting of processing modules (each module containing processor, cache, local memory) interconnected by a multistage interconnection network. Proteus provided us with accurate estimates of the execution times of the simulator on the multiprocessor, nonintrusive profiling, repeatability, means to measure the various overheads introduced by the method, and the capability to scale up the simulated multiprocessor and examine its impact on the behavior of our method. The accuracy of Proteus has been validated with several applications and multiprocessors [6] and, therefore, our experimental results are expected to closely match the executions of the simulations on real NUMA multiprocessors. We performed the same set of experiments as on the Alliant FX/80, and the results we obtained are shown in Figures 5 and 6.

In these experiments we examined four different implementations of SPaDES. They differ in the way we partition the simulated system across the processors (randomly (*rnd*) or by element strings (*str*)) and whether they utilize processor self-scheduling or static-scheduling (*Dyn* and *Stat* respectively). Partitioning by element strings is a two-step process. First, the system is divided into sets of *strings*, defined as sets of connected components with as most one fanin and one fanout included. Second, a load leveling

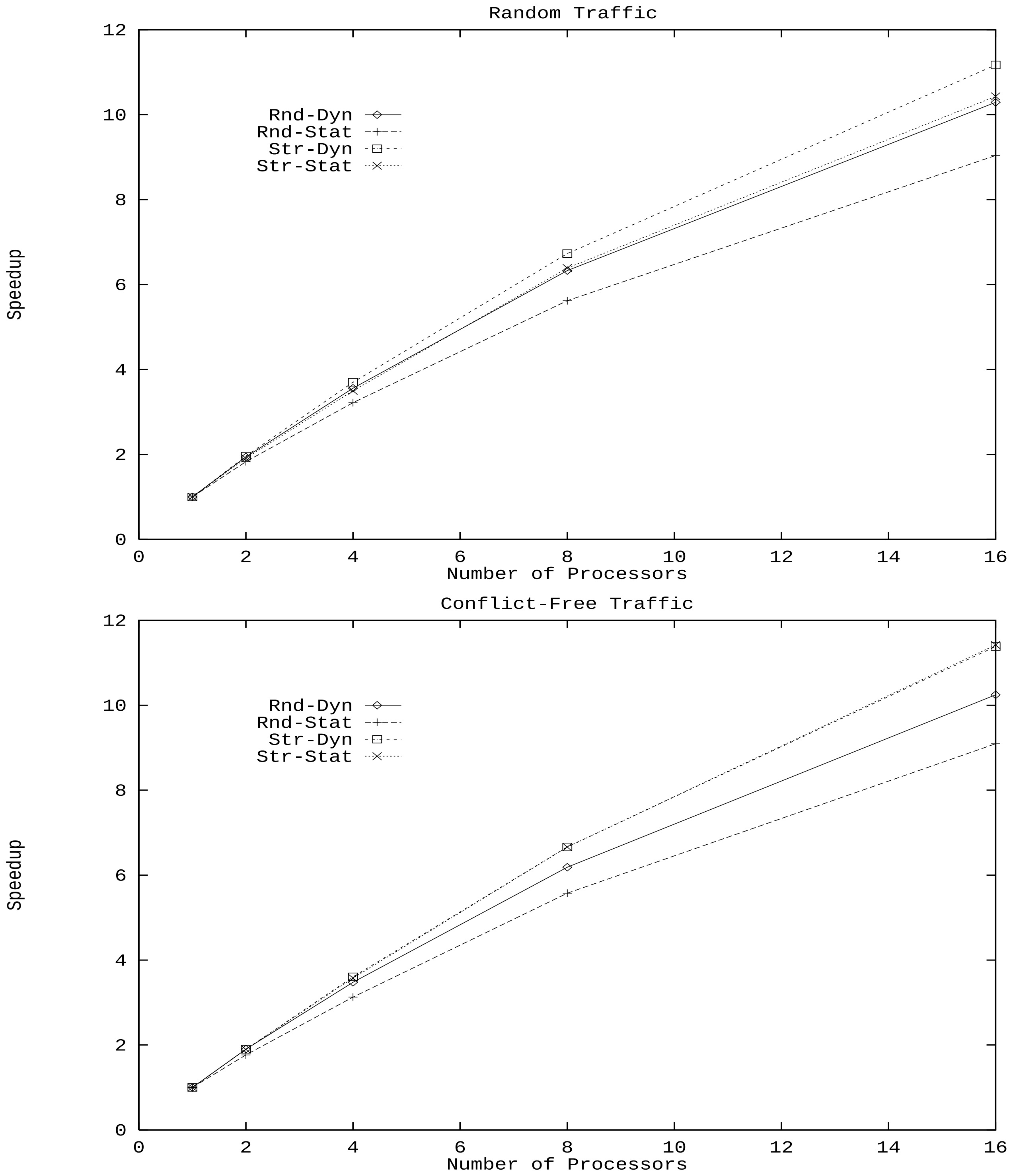
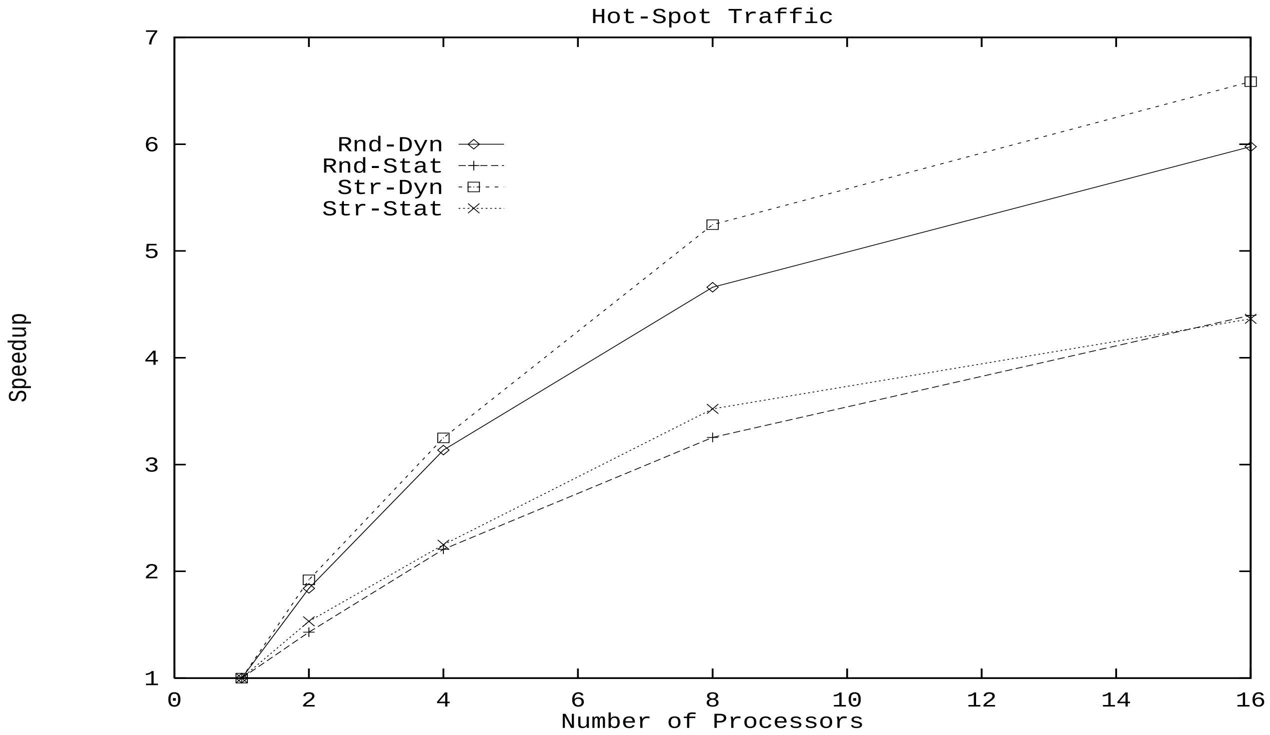


Figure 5: SPaDES Relative Speedups on a NUMA Machine



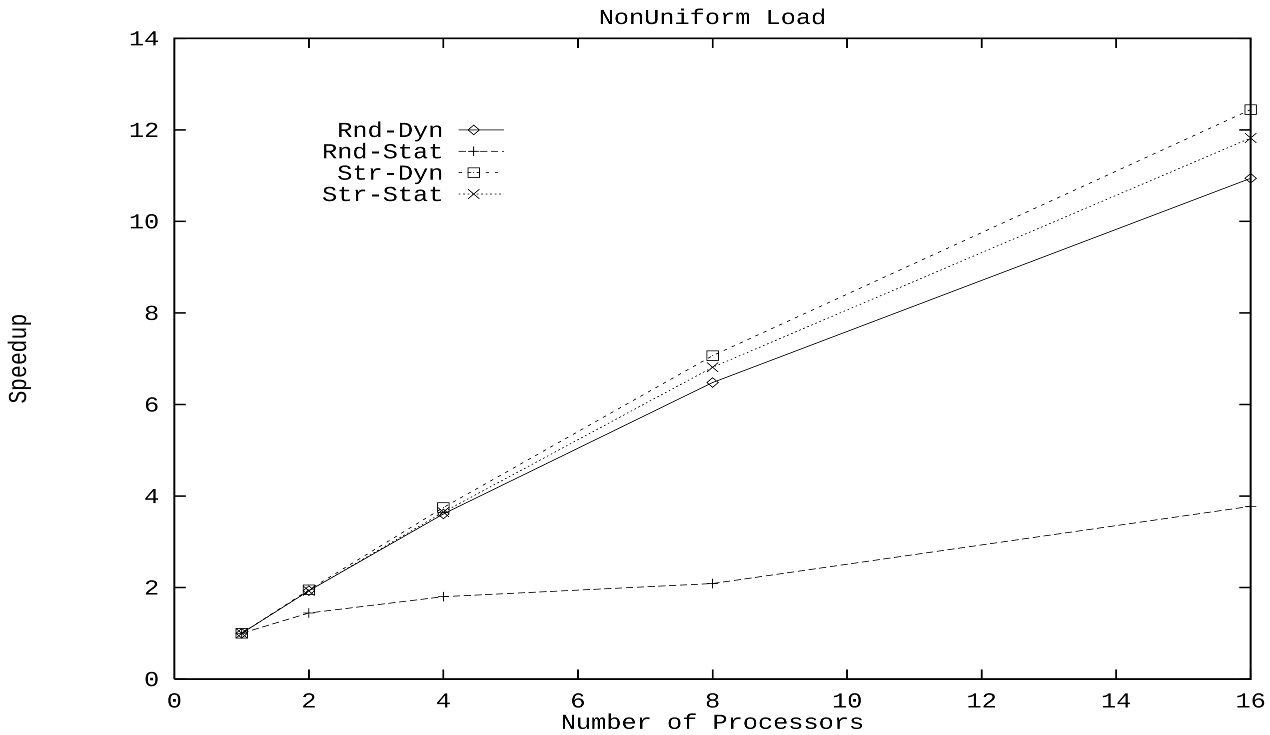


Figure 6: SPaDES Relative Speedups on a NUMA Machine

scheme is used to insure that block sizes are kept fairly close [1, 22, 34]. SPaDES performs well when there is sufficient amount of parallelism inherent in the simulation. It achieves a (relative) speedup of more than 11 on 16 processors (e.g. random and nonconflicting accesses simulations). The results also show the significant impact of additional issues, such as partitioning and processor self-scheduling, on the performance of the parallel simulator. Furthermore, the results show that a partitioning that performs well in one simulation (e.g. string partitioning in the nonconflicting accesses simulations) may not perform as well in another simulation with a different dynamic behavior (e.g. random partitioning in the nonuniform load simulations performs better than string partitioning). This unpredictability shows that partitioning alone seldom produces the most efficient simulator. Processor self-scheduling, on the other hand, can alleviate most of the negative impact of an unsuccessful partitioning (notice the improvement in the nonuniform load simulations employing random partitioning) and is necessary in obtaining the most efficient parallel simulator. Finally, the best relative performance of the method is achieved in the nonuniform load experiments, which are computation intensive simulations.

6 Conclusions

This paper describes a synchronous parallel discrete event driven simulation method (SPaDES) and its implementation on two shared memory multiprocessors. It also examines and compares its performance to Parsim, which is a parallel version of the traditional synchronous event-driven method.

The SPaDES method, which is based on the logical processes model, avoids the majority of the bottlenecks usually attributed to the parallel synchronous simulation methods and allows the efficient exploitation of the parallelism available in the simulations of synchronous systems. A multiprocessor system with an Omega interconnection network was simulated in order to study the performance of SPaDES

. The results obtained show that the performance as well as the efficiency of this method are high, and that this method achieves good speedups even with moderate amounts of parallelism. The few kernel operations introduced by SPaDES, the use of a single barrier during each simulation step, and the exploitation of a coarse-grain parallelism are the main reasons SPaDES achieves such a good performance and efficiency.

We can still improve the performance of SPaDES by lowering the overhead of several kernel operations, such as the event evaluation and the decision making process at the end of each step, and by adding a mechanism for aggressively exposing parallelism when there is not enough parallelism inherent in the simulation for the processors to exploit (aggressive SPaDES). As parallelism studies show [7, 36], the most striking characteristic of synchronous simulations is the clock effect: steps with high activity, followed by long relatively inactive periods. If a PDES method is to be efficient on such simulations it should handle well these long periods with limited parallelism. Unfortunately, the blocking barrier at the end of each step makes the performance of our synchronous method depend on the length of these periods. The longer the periods the worse the method will perform. On the other hand, we would like to avoid the overhead introduced by the parallelism-exposing mechanism when the simulation contains enough inherent parallelism to keep all the processors busy.

In this paper we examined the performance of the same simulator on two different shared memory multiprocessors. The two implementations differ only in the methods employed for partitioning and scheduling. The same simulator can be easily ported on other multiprocessors, such as KSR-1 and CM-5. We only need to implement the primitives for parallelism and synchronization in the simulator using the appropriate primitives provided by these machines. We are planning to examine the behavior and the performance of the simulator on these two multiprocessors and at the same time evaluate its portability on shared memory machines with different characteristics.

We need to continue our studies on parallel simulation methods and their performance in the simulation of synchronous systems. Two important applications which would benefit from synchronous methods are architectural and logic level designs, because such designs contain significant amounts of parallelism and are inherently synchronous. In addition, we need to extend the set of benchmarks we are using in our studies by examining logic level designs. Their efficient parallel simulation is difficult because they contain very fine grain parallelism which makes them sensitive to the overheads introduced by the simulation method. We also need to continue experimenting on shared memory multiprocessors because they provide the means for an efficient implementation of any simulation method.

References

[1] P. Agrawal. Circuit Partitioning for Hardware Simulation Engines. In *Proceedings of the 1985 International Conference on Computer Aided Design*, pages 161–163, 1985.

[2] R. Ayani. *Parallel Simulation on Shared Memory Multiprocessors*. PhD thesis, Dept. of Telecommunication and Computer Systems, The Royal Institute of Technology, Sweden, 1989.

[3] R. Ayani. A Parallel Simulation Scheme Based on Distances Between Objects. In *Proceedings of the SCS Western Multiconference on Distributed Simulation*, pages 113–118, March, 1989.

[4] R. Ayani and H. Rajaei. Parallel Simulation of a Generalized Cube Multistage Interconnection Network. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 60–63, January, 1990.

[5] B. Berkman and R. Ayani. Parallel Simulation of Multistage Interconnection Networks on a SIMD Computer. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, pages 133–140, January, 1991.

[6] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. In *Proceedings of the 1992 ACM SIGMETRICS and PERFORMANCE '92 Conference*, June 1992.

[7] J.V. Briner, Jr. *Parallel Mixed-Level Simulation of Digital Circuits Using Virtual Time*. PhD thesis, Duke University, August 1990.

[8] J. Bruner, C.J. Beckmann, P. Konas, D.K. Poulsen, and P.-C. Yew. An Environment for Parallel Execution-Driven and Event-Driven Architecture Simulation. *International Journal in Computer Simulation*, 1994.

[9] J. Bruner, H. Cheong, A. Veidenbaum, and P.-C. Yew. Chief: A Parallel Simulation Environment for Parallel Systems. In *Proceedings of the 5th International Parallel Processing Symposium*, pages 568–575, 1991.

[10] 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):440–452, September 1979.

[11] K.M. Chandy and R. Sherman. Space-Time and Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 53–57, 1989.

[12] K.M. Chandy and R. Sherman. The Conditional Event Approach to Distributed Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 93–99, 1989.

[13] R.M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):31–53, October, 1990.

[14] D. Gajski, D.J. Kuck, D.H. Lawrie, and A.H. Sameh. CEDAR – A Large Scale Multiprocessor. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 524–529, August, 1983.

[15] P.K. Goli, P. Heidelberger, D.F. Towsley, and Q. Yu. Processor Assignment and Synchronization in Parallel Simulation of Multistage Interconnection Networks. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 181–187, 1990.

[16] A. Gottlieb, R. Grishman, C. Kruskal, K. McAliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer. *IEEE Transaction on Computers*, C-32(2):175–190, February, 1983.

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

[18] P. Konas. Parallel Discrete Event Simulation on Shared Memory Multiprocessors. Technical Report CSRD, Rpt. No. 1074, Center for Supercomputing Research and Development, Master Thesis, Computer Science Department, University of Illinois, May, 1991.

[19] P. Konas and P.-C. Yew. Parallel Discrete Event Simulation on Shared-Memory Multiprocessors. In *Proceedings of the 24th Annual Simulation Symposium*, pages 134–148, April, 1991.

[20] P. Konas and P.-C. Yew. Synchronous Parallel Discrete Event Simulation on Shared-Memory Multiprocessors. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulations*, pages 20–32, January, 1992.

[21] D.H. Lawrie. Access and Alignment of Data in an Array Processor. *IEEE Transactions on Computers*, C-24(12):1145–1155, December, 1975.

[22] Y.H. Levendel, P.R. Menon, and S.H. Patel. Special Purpose Computer for Logic Simulation using Distributed Processing. *Bell System Technical Journal*, 61(10):2873–2909, December 1982.

[23] B.D. Lubachevsky. Bounded Lag Distributed Discrete Event Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 183–191, 1988.

[24] B.D. Lubachevsky. Scalability of the Bounded Lag Distributed Discrete Event Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 100–107, 1989.

[25] B.D. Lubachevsky. Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks. *Communications of the ACM*, 32(1):111–123, January, 1989.

[26] E.P. Markatos and T.J. LeBlanc. Memory-Conscious Scheduling in Shared-Memory Multiprocessors. Technical report, Computer Science Department, University of Rochester, 1991.

[27] E.P. Markatos and T.J. LeBlanc. Load Balancing vs Locality Management in Shared-Memory Multiprocessors. Technical Report 399, Computer Science Department, University of Rochester, October, 1991.

[28] W. Najjar, J.-L. Jezouin, and J.-L. Gaudiot. Parallel Discrete-Event Simulation. *IEEE Design and Test of Computers*, pages 41–44, December 1987.

[29] W. Najjar, J.-L. Jezouin, and J.-L. Gaudiot. Parallel Execution of Discrete-Event Simulation. In *Proceedings of the 1987 International Conference on Computer Design*, pages 668–671, 1987.

[30] D.M. Nicol. The Cost of Concervative Synchronization in Parallel Discrete Event Simulations. Technical Report 90-20, ICASE, May 1990.

[31] J. Palmer and Jr. G.L. Steele. Connection Machine Model CM-5 System Overview. In *Proceedings of Frontiers '92: Fourth Symposium on Massively Parallel Computation*, pages 474–483, 1992.

[32] R. Razdan, G. Bischoff, and E. Ulrich. Exploitation of Periodicity in Logic Simulation of Synchronous Systems. In *Proceedings of 1990 IEEE International Conference on Computer Aided Design*, pages 62–65, 1990.

[33] J. Rothnie. Overview of the KSR1 Computer System. Technical Report TR-9202001, Kendall Square Research, 1992.

[34] S.P. Smith, B. Underwood, and M.R. Mercer. An Analysis of Several Approaches to Circuit Simulation for Parallel Logic Simulation. In *Proceedings of the International Conference on Computer Design*, pages 664–667, 1987.

[35] L. Soulé and T. Blank. Parallel Logic Simulation on General Purpose Machines. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 166–171, 1988.

[36] L.P. Soulé. *Parallel Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms*. PhD thesis, Stanford University, June 1992.

[37] J.S. Steinman. Interactive SPEEDES. In *Proceedings of the 24rd Annual Simulation Symposium, 1991 Eastern Multiconference*, pages 149–158, 1991.

[38] J.S. Steinman. SPEEDES: Synchronous Parallel Environment for Emulation and Discrete. In *Proceedings of the SCS Multiconference on Distributed Simulation Event Simulation*, pages 95–103, 1991.

[39] E. Ulrich. A Design Verification Methodology Based on Concurrent Simulation and Clock Suppression. In *Proceedings of 20th Design Automation Conference*, pages 709–712, 1983.

[40] E.G. Ulrich. Exclusive Simulation of Activity in Digital Circuits. *Communications of the ACM*, 12(2):102–110, February, 1969.

[41] Q. Yu, D. Towsley, and P. Heidelberger. Time-Driven Parallel Simulation of Multistage Interconnection Networks. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 191–196, 1989.