PROCESSOR SELF-SCHEDULING IN PARALLEL DISCRETE EVENT SIMULATION

Pavlos Konas

Silicon Graphics Inc.
Mountain View, CA 94043, U.S.A.

Pen-Chung Yew

Department of Computer Science
University of Minnesota
Minneapolis, MN 55455-0159, U.S.A.

ABSTRACT

This paper describes a novel data structure and an algorithm for processor self-scheduling in parallel discrete event simulation. The presented data structure allows the efficient scheduling of future computations, it facilitates the inexpensive use of processor affinity information, it reduces the contention on the scheduling queue, and it integrates load balancing and locality management methods into a single mechanism. We use the behavioral simulation of a multiprocessor system to characterize the behavior of the proposed data structure and the associated scheduling algorithm. The results of our study show that it is important to maintain as detailed affinity information as possible and exploit this information at run time.

1 INTRODUCTION

In recent years, parallel discrete event simulation (PDES) has emerged as the principal technology which can satisfy the ever increasing demands for processing power and for storage of large simulations of architectural and logic-level designs. Synchronous PDES methods have shown significant potential in the parallel execution of such simulations (Konas 1994). On the other hand, asynchronous PDES methods (both conservative and optimistic) are less attractive due to the significant overheads they introduce in attempting to find parallelism during a simulation (Konas and Yew 1991).

The performance of a parallel simulation depends on the efficient utilization of the processors in a multiprocessor system. Two significant factors that determine the effective use of a multiprocessor are load balancing and locality management (Markatos 1993). *Load balancing* refers to the dynamic redistribution of the workload among the participating processors so that the load is continuously balanced across the multiprocessor. *Locality management*, on the other hand, refers to the execution of a computation on the processor closer to the storage where the data associated with this computation has been allocated.

Even though both load balancing and locality management methods attempt to improve the performance of a parallel program, their optimization goals are conflicting. Thus, we need to balance the potential costs and benefits of both load balancing and locality management approaches in order to achieve a highly efficient utilization of the parallel system. Ignoring either factor may cause significant performance degradation to the parallel program.

One way to account for these two factors is by partitioning the simulation across the processors of the parallel machine. Unfortunately, partitioning (also known as static scheduling) cannot predict the dynamic behavior of a simulation, and frequently results in ill-suited assignments of computations to processors and of data to memories. Processor self-scheduling can be used to dynamically determine how to execute most effectively the parallel simulation on a multiprocessor.

In a synchronous PDES simulation processor self-scheduling is critical to achieving an efficient parallel execution because of several factors. First, the synchronous nature of the parallel method makes the performance of such a simulator very sensitive to load imbalances during any of the simulation steps. Second, partitioning is usually unable to predict and account for the dynamic behavior of the parallel execution of a simulation because of the variability of the components' simulation execution times, and the transient nature of the activity in the simulated system. Third, most of the computations in architectural and logic-level simulations are fine- to medium-grain computations and, therefore, long memory access delays can introduce significant overheads into the parallel execution. Finally, in simulations based on the logical processes (LP) model (Fujimoto 1990) the distribution of the event queue across the LPs makes the scheduling queue(s) the only mechanism available for transforming conditionally activated LPs (scheduled computations) into unconditionally active LPs (ready to execute computations).

In this paper we present a novel data structure and an algorithm for processor self-scheduling in synchronous PDES simulation. The presented data structure offers several advantages. It allows the efficient scheduling of potential future computations. It facilitates the inexpensive use of processor affinity information in allocating processors to active LPs. It reduces the probability of contention on the scheduling queues. It integrates load balancing and locality management methods into a single mechanism. Finally, deciding the next simulation step at the end of the current simulation step can be performed very efficiently.

The main concern in this paper is the presentation of an approach to processor self-scheduling in synchronous PDES simulations. In Section 2 we present a data structure and an algorithm for processor self-scheduling in parallel simulations. In Section 3 we examine the performance of our approach compared to a simpler data structure containing less affinity information. Finally, Section 4 summarizes the work presented in this paper, and proposes interesting extensions to this research.

2 A PROCESSOR SELF-SCHEDULING MECHANISM

Processor self-scheduling is critical to the performance of synchronous PDES simulations because of the synchronous nature of the parallel method; the small grain sizes of the computations; the transient nature of the activity in the simulated system; and the functionality of the scheduling queue as the only mechanism for transforming conditionally activated LPs to unconditionally active LPs. Thus, we need to carefully design the scheduling queue data structure as well as the algorithm for allocating idle processors to the execution of active LPs.

In a synchronous PDES simulator, processor self-scheduling has two functionalities. First, it assumes the functionality of the event queue in traditional event driven simulations: it is a repository for conditionally activated LPs, and the only mechanism for transforming these *conditional activations* into *unconditionally active* LPs. The two main reasons for the scheduling queue assuming this functionality in a synchronous PDES simulator are the absence of a centralized event queue, and the incapability of an LP to decide on its own whether it is conditionally activated or it has already become unconditionally active. In order to reduce the overheads introduced by the parallel simulation method, a synchronous PDES method strips the LPs from all the control (decision) sequences. Thus, the LPs only know how to simulate the corresponding physical processes when they are scheduled to do so.

Therefore, we need a scheduling queue data structure that will be efficient in scheduling conditionally activated LPs for future consideration, as well as assigning unconditionally active LPs to idle processors, in much the same way as an event queue handles events in a traditional event driven simulation. Previous work in the simulation area has shown that the most efficient data structure for event scheduling in simulations of detailed logic networks of large and active digital systems is the *time-wheel* (Razdan, Bischoff, and Ulrich 1990, Ulrich 1980, Ulrich 1983).

The time-wheel actually consists of two data structures: a collection of linked lists containing scheduled future events (potential future activity in the simulated system), and a small hash table that maps the simulated time of an event occurrence to one of the linked lists. Analysis of this event queue data structure has shown that it can maintain an $O(1)$ average performance in both its tasks (scheduling future computations, and allocating computations for execution on idle processors). However, the performance offered by the time-wheel data structure quickly degrades when events are scheduled on the overflow list.

In practice, the time wheel is (almost) guaranteed to have an average performance of $O(1)$ in the simulation of architectural and logic-level designs. The reason is the behavior of these types of simulations. As has been observed in (Konas 1994, Soulé 1992), the most striking characteristic of architectural and logic-level simulations is the "clock" effect: peaks with considerable activity induced by the arrivals of the clock and of new input signals, followed by periods of diminishing activity as the signals propagate through the combinational elements of the simulated system. Thus, a scheduling queue implemented as a time-wheel that covers a simulated time interval equal to the clock period or to the propagation delay through a combinational circuit should be expected to achieve a constant $O(1)$ access time. Furthermore, the components of architectural and logic-level designs usually assume delays from a small set of values. In this case, a time-wheel that handles well this small set of values introduces minimum overhead in scheduling future computations, and thus it is ideal for the parallel simulation of such designs.

The second functionality of processor self-scheduling in a synchronous PDES simulation is similar to that of the scheduling pool in self-scheduled parallel programs: it is a pool containing ready-to-execute computations which need to be assigned to idle processors in such a way that the total execution time of the parallel program is minimized. Previous work in the area of self-scheduling of parallel programs has shown that, in the presence of nonuniform memory access (NUMA) characteristics in the host multiprocessor, we need to simultaneously address the issues of load balancing and of locality management if we are to achieve the most efficient parallel execution (Markatos 1993). It has also been shown that scheduling decisions should be fast introducing minimum overhead into the computation (Anderson 1991). This means that affinity related decisions should use minimum information, and should not introduce significant overhead into the scheduling process. Otherwise, the performance of the parallel program would suffer (Anderson 1991). Furthermore, both analytical and experimental results have shown that a single, centralized scheduling queue will quickly become the system bottleneck as the number of participating processors increases (Squillante 1990).

The consensus of the previous work in scheduling thread-based and loop-parallel programs has been to avoid single, centralized data structures, and use affinity information when available. Most of the studies in this area propose the use of per-processor scheduling queues together with scheduling algorithms that deposit ready-to-execute computations into the other processors' queues, or search the other processors' queues for ready computations, or do both. A combination of per-processor data structures with a centralized queue has also been proposed in order to minimize the overhead associated with searching all the other processors' scheduling queues for ready-to-execute computations. According to the above observations, we need a scheduling queue per processor which will provide easy to use affinity information.

Since processor self-scheduling in a synchronous PDES simulation assumes simultaneously the functionalities of both an event queue and of a scheduling pool, and the performance of the parallel simulator critically depends on how well it performs in both these tasks, we should design a data structure and a scheduling algorithm that combine data structures and algorithms known to perform well in each of the two areas. A time-wheel scheduling queue augmented with affinity information seems to be the most appropriate approach to processor self-scheduling in a synchronous PDES simulator.

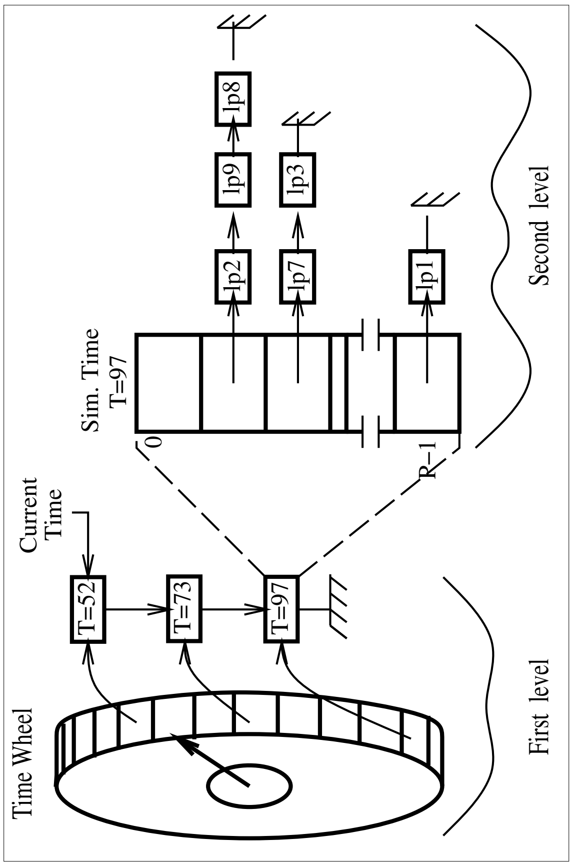

Figure 1: The Scheduling Queue Data Structure

2.1 The Scheduling Queue Data Structure

Figure 1 shows a two-level data structure that combines the concepts of the time-wheel and of affinity scheduling. The first level of the data structure is a variation of the time-wheel. It consists of a hash table which maps the simulated time of an LP's activation to a node of a linear linked list. Each node of this list contains all LPs activated for a particular simulated time. Using faster but more complicated data structures than the linear linked list is not necessary, since the structure is updated only twice per simulation step. First, a new node is *inserted* into the list when an LP is activated for a simulated time that is not already present in the linked list. All LPs subsequently scheduled for that simulated time will be directly scheduled into the existing node through the time-wheel entry. Second, a node is *deleted* from the list when all LPs scheduled for the current simulation time have been simulated; that is, upon completing the execution of a simulation step the node representing the step is deleted from the scheduling queue. The first level of the presented data structure provides us with the fast access times guaranteed by the time-wheel in the simulation of architectural and logic-level designs.

The second level of the scheduling queue data structure provides "cheap" affinity information that can be easily exploited during the execution of a simulation. Each node of the linked list consists of a P-entry array, where P is the number of processors participating in the parallel execution of the simulation. Each entry of this array points to a linked list which contains all the LPs activated for that simulated time and which (LPs) have been assigned during a pre-simulation partitioning to the processor corresponding to that entry (host processor of the LPs). The idea behind the P-entry array is to allow a processor to simulate first all active LPs whose corresponding data structures have been assigned to the local memory of the processor, before it starts simulating LPs whose corresponding data structures are located in remote memory modules. In this way preference is given to locality management but load balancing is also taken into consideration.

In synchronous PDES simulation, a logical process is activated when a new value arrives at its inputs and no event already exists in the input queue of the LP for that simulated time (Konas 1994). The activation of an LP results into its insertion in the scheduling queue of the processor that activated it. More specifically, when a processor activates an LP it executes the following steps. First, the processor finds the node in its local queue corresponding to the simulated time of the LP's activation. If such a node does not exist in the scheduling queue, the processor creates a new node, inserts it into the list, and updates the appropriate slot of the time-wheel. Then, the processor inserts the LP activation record into the entry corresponding to the host processor of that LP. In this way, when a processor checks the scheduling queue for ready-to-execute LPs, it simulates first LPs whose associated data is stored locally, and then, when it can find no such processes, it simulates LPs whose data is stored in remote memory modules.

This scheduling queue organization provides cheap object-based affinity information: the affinity of an LP toward a processor is represented by the location of its activation record in the scheduling queue node. It can also be easily exploited: a processor can use the affinity information by just accessing the appropriate entry in the scheduling queue node representing the current simulation time. Instead of assigning an LP to the entry corresponding to its host processor, we could assign it to the entry corresponding to the processor on which the LP was last executed. This is a form of thread-based affinity, since the affinity depends on the last execution location of the LP. However, experimentation has shown that there is no advantage in using thread-based affinity over object-based affinity, since very rarely the footprint of an LP would remain on a processor's cache for long after the simulation of the LP's activation has been completed.

An interesting implementation issue associated with a scheduling queue is the allocation and recycling of nodes of the scheduling queues, and of LP activation records. In which memory module does a processor allocate these objects? Obviously, a centralized memory handler would soon become a bottleneck in a parallel simulation. In a NUMA environment which allows the user to control the module on which memory is allocated, per processor memory handlers facilitate fast and inexpensive memory manipulation. However, distributed memory handlers have an important consequence: it is no longer profitable to schedule LPs directly onto other processors' scheduling queues. Thus, a processor schedules LPs only on its local scheduling queue. Under such conditions and due to the importance of the affinity concept, a processor needs to know which of the LPs whose data is stored locally have been activated remotely for the current simulation step. Using the data structure shown in Figure 1, a processor can easily check whether such LPs exist and on which scheduling queues they have been scheduled. Based on this information, the processor can access the appropriate queues and acquire its local LPs for simulation.

2.2 The Scheduling Algorithm

So far we have described a data structure to be used as the scheduling queue in a synchronous PDES simulation. Now we need to provide a scheduling algorithm which efficiently utilizes this data structure. A scheduling algorithm has two responsibilities: to schedule a conditionally activated LP for future execution, and to allocate an available (idle) processor to an unconditionally active LP. The first responsibility is incorporated into the LPs. As we described earlier, whenever a new value arrives at the inputs of an LP, the LP is inserted into the scheduling queue for future consideration.

On the other hand, the allocation of an idle processor to an unconditionally active LP in such a way that the execution time of the parallel simulation is minimized is the responsibility of the scheduler. When a synchronous PDES simulation starts executing, it forks a scheduler process on each participating processor in the parallel machine. The task of the scheduler is to multiplex the execution of active LPs on the corresponding processor until the simulation is completed. Then it joins the other schedulers in terminating the parallel execution of the simulation. This type of execution is known as the work-pool model.

Each scheduler executes the algorithm shown in Figure 2. During each simulation step, a scheduler first simulates the *local LPs* which are scheduled on its *local queue*. In this way, we exploit the available affinity information to achieve the most efficient execution of these LPs. When no more local LPs remain on the local queue, the scheduler searches the other processors' scheduling queues for *local LPs* which have been *scheduled remotely*. This step also aims in exploiting object-affinity to produce an efficient execution of active LPs. Finally, the scheduler simulates any remaining active LPs scheduled for the current simulation time. The search for such remaining LPs starts at the processor's local queue and visits all the scheduling queues in a round robin fashion. This last step provides load balancing capabilities to the scheduling algorithm. Notice, however, that preference is given to exploiting affinity, and that load balancing becomes an issue only when there are no more active LPs with affinity toward the particular processor.

The algorithm, however, contains a significant overhead hidden in its last step. Since each processor searches all P entries of a scheduling queue node, and does that for all the scheduling queues, the potential cost of the load balancing step of the scheduling algorithm is $O(P^2)$.

This effect of this overhead is shown graphically in Figure 3. These graphs show the execution time of a behavioral parallel simulation of a shared-memory multiprocessor (simulated system) as we increase the number of processors on the host machine. Three different traffic models are used in these simulations, each produc-

```
STEP-1:
// local LPs, scheduled locally
while there are local active LPs in my queue
    acquire and delete the first such LP
    simulate the LP for the current time
end while
```

```
STEP-2:
// local LPs, scheduled remotely
For each other scheduling queue
    find the entry corresponding to my ID
    while there are active LPs in the entry
        acquire and delete the head of the queue
        simulate the LP for the current time
    end while
end for
```

```
STEP-3:
// remote LPs, scheduled both locally and remotely
For each scheduling queue starting with my own
    For each entry containing activated LPs
        while there are active LPs in the entry
            acquire and delete the first such LP
            simulate the LP for the current time
        end while
    end for
end for
```

Figure 2: The Scheduling Algorithm

ing a different amount of activity during the parallel simulation (low in hot-spot traffic, medium in random traffic, and high in conflict-free traffic). A fourth simulation (nonuniform load) introduces nonuniformity in the computation requirements of the simulated components. In all simulations we pre-partition the system using both random (RND) and string-based (STR) partitioning methods (Konas 1994), and turn self-scheduling on (DYNamic) and off (STATic). The overhead introduced into the computation by the load balancing step is (most of) the difference between the static and dynamic versions of each simulator. Bit flags could be used to reduce the scheduling overhead to acceptable levels in all practical situations. However, this hidden overhead could potentially produce an unstable behavior of the scheduling algorithm.

Fortunately, there is a way to avoid such an overhead. The solution is based on the concept of combining trees, on the existence of the barrier synchronization at the end of each simulation step, and on the shared memory of the host multiprocessor. In a synchronous PDES simulator, where a barrier synchronizes the processors at the end of each simulation step, the quadratic scheduling overhead problem is solved as follows. As each processor partici-

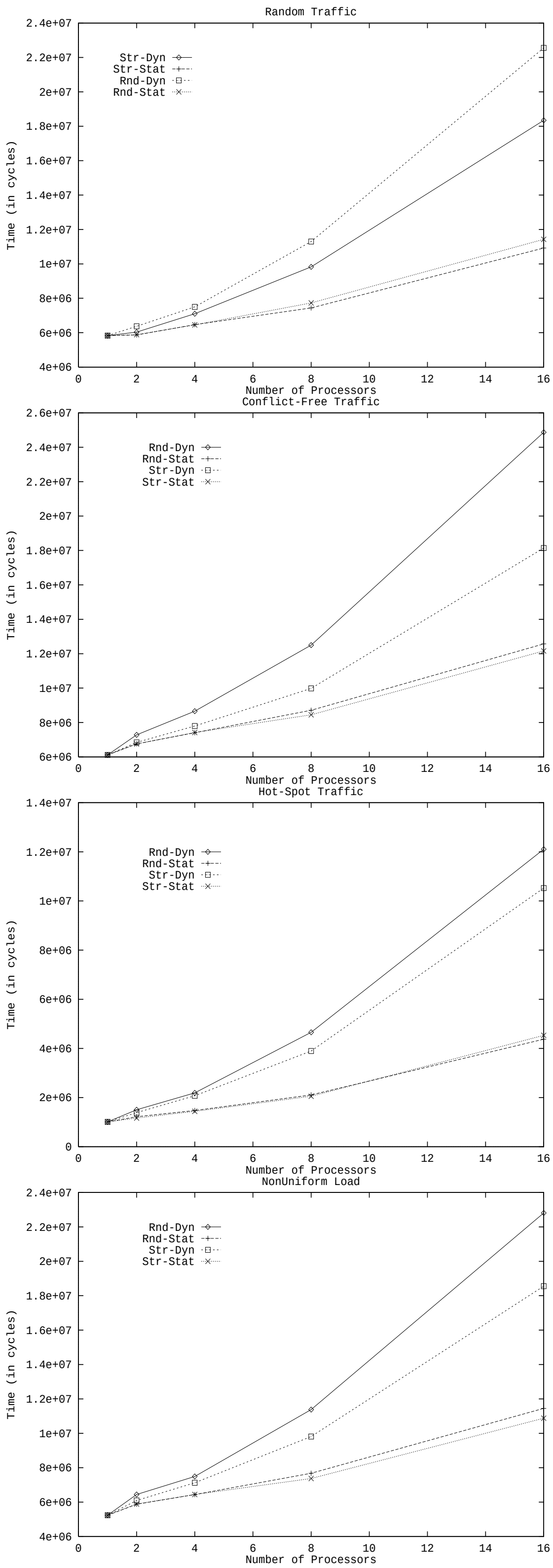

Figure 3: Overhead of the Scheduling Method

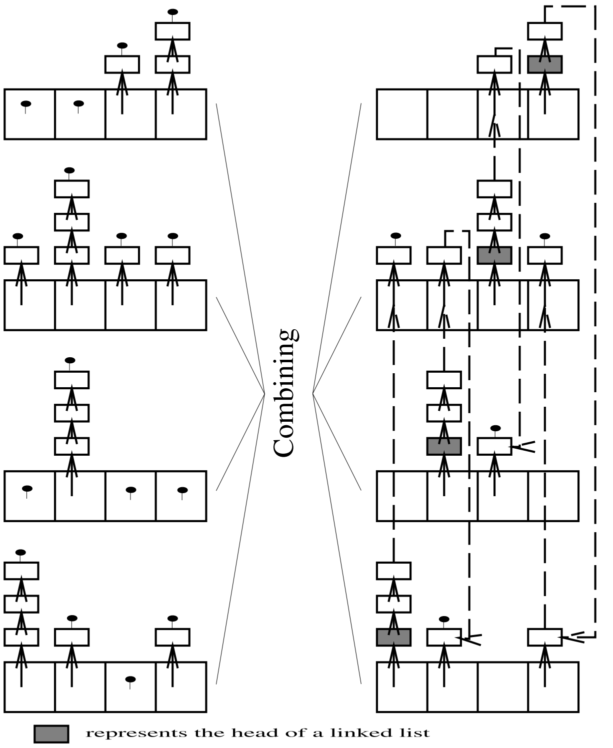

Figure 4: Combining of Scheduling Queue Entries

pates in the barrier, it combines the entries of its scheduling queue node representing the next simulation step with the nodes of the other processors representing the same simulation step. When the barrier is completed, the entries corresponding to a single processor in all the scheduling queue nodes representing the current simulation step, are chained together forming a linked list. The head of this list coincides with the corresponding entry in the local queue of that processor. Thus, a processor can simulate all the *local LPs* which have been scheduled both *locally* and *remotely* by just following the links in the constructed list. In addition, a processor only needs to check the entry corresponding to another processor on the other processor's scheduling queue in order to gain access to all the LPs local to that processor which are still waiting to be simulated for the current simulation time. Notice, that with this optimization, the second step of the scheduling algorithm no longer exists since it has been integrated into the first step. The important result of this optimization is that the scheduling overhead becomes linearly dependent on the number of processors participating in the parallel execution of the simulation. Figure 4 shows a simple example of this combining process. The combining of the nodes of these four scheduling queues results in the creation of four linked lists, each of which contains the LPs local to each of the four processors participating in the simulation.

3 PERFORMANCE STUDY OF THE SCHEDULING APPROACH

In order to study the performance of the presented data structure and scheduling algorithm, and evaluate the usefulness of the affinity information maintained by this structure, we performed the experiments described in the previous section but with the combining optimization in place. In this study, however, we compare the performance of the simulator using the presented scheduling queue data structure to a simulator using a similar but much simpler scheduling queue data structure. Since the usefulness of the time-wheel data structure has been previously demonstrated, we do not examine alternative data structures for the first level of the scheduling queue. However, the second level of the data structure might be too complicated for the performance it provides. Therefore, we examine a simpler data structure where the second level of the structure is a two-entry array (instead of a P-entry array). The first entry is dedicated to scheduling local LPs, whereas the second entry is for scheduling remote LPs. As a result of this organization, a processor cannot easily know about and access its local LPs scheduled remotely. Thus, the second step of the scheduling algorithm shown in Figure 2 is not needed with this simpler data structure.

The results of our experiments are shown in Figure 5. We observe that the performances of the two simulators differ significantly, and this difference widens as the number of processors executing the simulations increases. In addition, simulations that contain more activity (e.g., random traffic and nonconflicting accesses) result in a large difference in the respective performances of the simulators. This means that in larger systems the presented data structure is expected to perform significantly better than the simpler structure. Furthermore, in simulations where the simulation method does not perform so well (e.g., hot-spot traffic), the scheduling method may have an additional negative impact on the performance of the simulator if it does not take full advantage of the affinity information.

The results of this brief study show that maintaining as detailed affinity information as possible, and exploiting it at run time, improves the performance of the parallel simulation (compared to simulators utilizing a smaller amount of affinity information). Thus, the complexity of the presented data structure is justified by the performance improvement it provides over simpler data structures which maintain a smaller amount of affinity information.

4 CONCLUSIONS

In this paper we presented a data structure and an algorithm for processor self-scheduling in a synchronous PDES simulation. By viewing a synchronous parallel simulator as a parallel program, we argued that there are

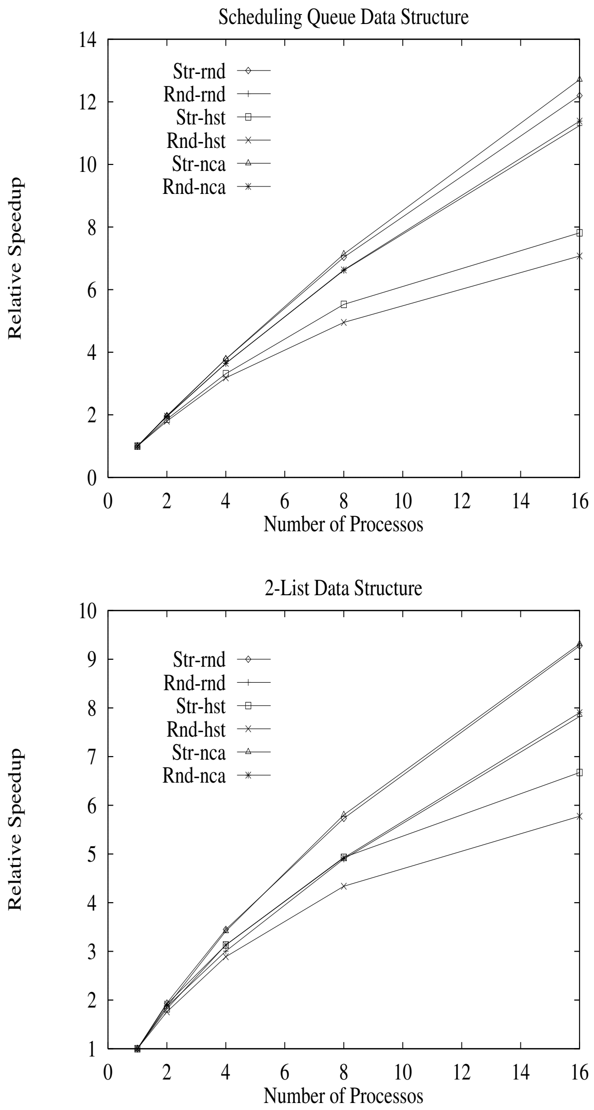

Figure 5: Performance of the Scheduling Approach

two significant factors that determine its performance: load balancing and locality management. Communication overheads induced by poor placement of data or of computations, and idle processors resulting from load imbalances during any simulation step, can cause significant degradation to the performance of a synchronous PDES simulator.

Processor self-scheduling in a synchronous PDES simulation assumes both the functionality of an event queue of a traditional event driven simulation, and the functionality of a work pool in self-scheduled parallel programs. We presented a two-level data structure that accounts for both functionalities.

The first level, which is a variation of the time-wheel, exploits characteristics of the behavior of architectural and logic-level designs to achieve an almost constant scheduling time of conditionally activated LPs. The second level provides inexpensive affinity information that can be easily exploited. In addition, it facilitates the integration of locality management and load balancing methods into a single mechanism aiming at the most efficient execution of a synchronous PDES simulation.

We also presented a scheduling algorithm which efficiently utilizes the presented data structure. The algorithm guides the allocation of idle processors to unconditionally active LPs during the execution of the simulation. During each simulation step, a processor first executes all local active LPs which have been scheduled locally or remotely. Then it considers for execution any remote LPs which are still waiting to be simulated. Thus, the algorithm first tries to exploit the affinity of processes toward processors, and addresses the load balancing issue only when there is no more affinity information that can be exploited.

The work presented here can be extended in two ways. First, it would be interesting to study the behavior and the performance of the presented data structure using analytical models, such as the Hold and the p-Hold (Riboe 1990) models. These models allow us to study a data structure under different future event scheduling distributions, and have been extensively used in comparing different event queue implementations (Chou, Bruell, and Jones 1993). Such a study will provide us with a more general characterization of the suitability of the presented data structure as a scheduling queue in general-purpose simulations. Second, we need to study the performance of diverse parallel simulators using this data structure as their scheduling queue. It would be interesting to exercise the data structure in simulations with different characteristics such as synchronous logic-level simulations, combinational circuit simulations, and architectural design simulations.

Processor self-scheduling is very important to the efficient parallel execution of simulations, especially in synchronous parallel simulations. Data structures and algorithms should be carefully designed so as to not introduce unnecessary overheads. They should also exploit simulation-specific characteristics in order to improve the efficiency of the resulting parallel simulator.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Grant Nos. MIP 93-07910 and MIP 94-96320, NASA NCC 2-559, and a donation from Motorola.

REFERENCES

Anderson, T. 1991. Operating System Support for High Performance Multiprocessing. PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington.

Chou, C., Bruell, S., and Jones, D. 1993. A Generalized Hold Model. In *Proceedings of the 1993 Winter Simulation Conference*, 756–761.

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

Konas, P. 1994. Parallel Architectural Simulations on Shared-Memory Multiprocessors. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

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

Markatos, E. 1993. Scheduling for Locality in Shared-Memory Multiprocessors. PhD thesis, Department of Computer Science, University of Rochester, Rochester, New York.

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

Riboe, J. 1990. Improvement of the Calendar Queue Algorithm. Technical report, Department of Telecommunication and Computer Systems, The Royal Institute of Technology, Stockholm, Sweden.

Soulé, L. 1992. Parallel Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms. PhD thesis, Stanford University, Palo Alto, California.

Squillante, M. 1990. Issues in Shared-Memory Multiprocessor Scheduling: A Performance Evaluation. PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington.

Ulrich, E. 1980. Table Lookup Techniques for Fast and Flexible Digital Logic Simulation. In *Proceedings of the 17th Design Automation Conference*, 560–563.

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

AUTHOR BIOGRAPHIES

PAVLOS KONAS received the B.S. (Ptychion) in Computer Science from the University of Crete-Greece in 1988, the M.S. and Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 1991 and 1994 respectively. He is currently engaged in the development of CAD tools at Silicon Graphics, Inc. His research interests include parallel simulation, parallel architectures, performance evaluation, object-oriented design, and parallel processing.

PEN-CHUNG YEW is a full professor in the Department of Computer Science, University of Minnesota since 1994. Before that he was an associate director of the Center for Supercomputing Research and Development, Univesity of Illinois at Urbana-Champaign. He received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign. From 1991 to 1992, he served as the Program Director of the Microelectronic Systems Architecture Program in the Division of Microelectronic Information Processing Systems at the National Science Foundation, Washington, D.C.. He is on the editorial board of the IEEE Trans. on Parallel and Distributed Systems. His current research interests include parallelizing compiler, shared-memory multiprocessor system design and parallel simulation.