Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

AUTOMATIC PARALLELIZATION OF DISCRETE EVENT SIMULATION PROGRAMS

Jya-Jang Tsai
Richard M. Fujimoto

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

ABSTRACT

Developing parallel discrete event simulation code is currently very time-consuming and requires a high level of expertise. Few tools, if any, exist to aid conversion of existing sequential simulation programs to efficient parallel code. Traditional approaches to automatic parallelization, as used in many parallelizing compilers, are not well-suited for this application because of the irregular, data dependent nature of discrete event simulation computations.

In this paper, we present an approach for automatically parallelizing sequential discrete event simulation (DES) programs. A framework for parallel execution is developed into which sequential DES languages are mapped. Using a Time Warp like execution mechanism, we demonstrate that it is possible to automatically translate DES programs into equivalent parallel programs. A case study of the SIMSCRIPT II.5 language is conducted, and preliminary performance data based on a prototype parallelizing SIMSCRIPT compiler and run-time system are presented.

1 INTRODUCTION

Discrete event simulation (DES) has long been a widely used technique for analyzing complex dynamical systems such as communication networks, computer systems, and combat scenarios, to mention a few. These applications often require excessively long execution times on sequential machines. As multiprocessor computers become more widespread, parallel execution of discrete event simulation programs (PDES) offers great potential for significantly reducing model execution time.

However, developing parallel simulation software is not only time-consuming but also requires much more training and experience than developing sequential software. Further, in many cases, it is difficult, if not impossible, for users to abandon large sequential simulation programs that they have been developing and using for years. It is clear that there would be enormous benefit if these sequential programs could be readily translated into efficient parallel programs.

In this paper, we describe a method to *automatically* translate a sequential DES program into an equivalent parallel version. In the following, we first review related work. Our framework for automatic parallelization that includes a common set of simulation primitives is presented, followed by a description of the approach used to parallelize these primitives. A case study examination of the SIMSCRIPT II.5 language is presented, and a prototype implementation is described. Finally, we describe preliminary performance data illustrating the potential speedup that is possible for these applications, and the actual speedup that is realized on a Kendall Square Research KSR-1 multiprocessor.

2 RELATED WORK

Much of the work in PDES has been concerned with synchronizing the simulation computation, and a number of protocols have been proposed [Fujimoto, 1990]. These approaches usually assume that the simulation program is written "from scratch" for parallel execution, and do not consider the question of utilizing existing sequential simulation programs.

Optimistic synchronization protocols such as Time Warp [Jefferson, 1985] have demonstrated good performance in a variety of applications [Fujimoto, 1990]. Unlike so-called conservative protocols that avoid synchronization errors, Time Warp detects out-of-order event execution at run-time, and uses a rollback mechanism to recover. Here, a Time Warp like execution mechanism that includes support for shared state variables [Fujimoto, 1989] is used.

Some approaches have been proposed that allow concurrent execution of *sequential* simulation programs. The *replicated trials* approach executes independent copies of the simulation program on separate processors. This is a suitable approach when collecting data over a wide range of parameter settings, or when long simulation runs are needed to reduce the variance of output statistics. This is normally the method of choice in those situations, although memory requirements may be large because N times the memory of a sequential execution is required when using N processors.

The *functional specialization* approach [Comfort, 1991, Paterra *et al.*, 1990] also does not require modifications to the sequential simulation program. Dedicated processors are used to execute frequently used functions such as random number generation and event list manipulation. However, this approach only offers a limited amount of speedup.

Much research has been completed concerning techniques to identify data dependence relationships between instructions in order to parallelize arbitrary sequential programs [Polychronopoulos, 1988, Szymanski, 1991, Wolfe, 1988]. These techniques typically focus on concurrent execution of iterations of DO-loops. This approach has difficulty in analyzing DES programs, however, because dependence constraints are usually complex, and difficult to predict at *run-time*, let alone compile-time. Nevertheless, as demonstrated later, program restructuring techniques can be fruitfully used in conjunction with the approach proposed here.

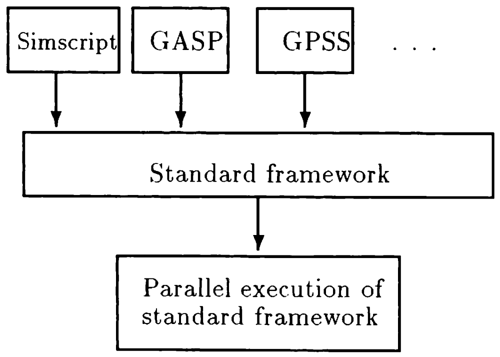

Figure 1: Relationship between standard framework and sequential simulation languages

3 A FRAMEWORK FOR DES

In our parallelization system, a common framework is first defined to which sequential discrete event simulation programs are mapped (see Figure 1). In our framework a DES program maintains two sets – a set of state variables S, and a set of unprocessed events, E. The former is used to describe the *system state* of the model, and the latter to specify state changes that will occur in the simulated future. During the execution of an event, one or more of the following four operations may be performed on the state variable set: (1) create a new variable (i.e., $S' = NS(S, v) = S \cup \{v\}$, where v is a variable), (2) read the contents of a variable (i.e., $v_0 = RS(S_{v/v_0}, v)$, S_{v/v_0} means that S includes a variable v whose value is v_0), (3) modify the contents of a variable (i.e., $S_{v/v_0'} = MS(S_{v/v_0}, v)$), and (4) delete a variable (i.e., $S' = DS(S, v) = S - \{v\}$). Similarly, three operations may be performed on the event set: (1) add a new event (i.e., $E' = SE(E, e) = E \cup \{e\}$ where e is an event to be scheduled, represented by a tuple containing the function to be executed, its arguments, and the simulated time at which event is to be executed), (2) delete or unschedule a previously scheduled event (i.e., $E' = DE(E, e) = E - \{e\}$ where e is an event to be unscheduled), and (3) examine the contents of the event set and return some value based on the event set, for example, the number of pending events (i.e., $V = EE(E)$ where V could be any value, e.g., boolean, integer, or a set of events). The execution of a sequential DES program can be viewed as a sequence of transitions from one $< S, E >$ pair to another.

These seven primitives (i.e., **NS**, **RS**, **MS**, **DS**, **SE**, **DE** and **EE**) appear to be sufficient to model a variety of existing simulators written in simulation languages, such as GASP, GPSS, and SIMSCRIPT II.5. Since higher level constructs (e.g., processes) can be realized as abstractions that are built on top of these primitives, these primitives are able to encompass applications with different paradigms of modeling, or *world views* [Kay, 1972, Kiviat, 1967, Pritsker, 1974, Spriet and Vansteenkiste, 1982, Zeigler, 1976].

4 CONCURRENT EXECUTION OF THE SIMULATION PRIMITIVES

Here, we assume the parallelized program executes on a shared memory multiprocessor. The execution mechanism is a simple generalization of the event list mechanism used by sequential simulators. Specifically, each processor repeatedly removes the smallest timestamped event from the event list, and then executes simulation code to process that event. The simulation code is functionally similar to that of a sequential simulator. The run-time system ensures that the parallel execution achieves the same results as the sequential execution.

Two critical issues must be addressed before the above approach to parallelization can provide the same results as the sequential execution. First, two concurrently executing events, each containing a different timestamp, may access a common state variable. This is not allowed in most PDES protocols (e.g., Time Warp). Consider two events, one with timestamp 10 and the other with timestamp 20, that both access a state variable X. Should X hold the value that existed at simulated time 10, or the value at time 20? To address this question, we use an abstraction called Space Time Memory (STM) [Fujimoto, 1989, Ghosh and Fujimoto, 1991] and a computation model based on optimistic synchronization.

The second problem concerns the EE primitive that accesses the event list. At any point in time, the event list in the parallel execution will be substantially different from that of the sequential execution, and may be distributed across several processors. Later, we suggest a possible solution to this problem.

4.1 Space Time Memory

The optimistic execution mechanism processes events containing different timestamps concurrently. Since events with different timestamps may have different views of the system state, it is imperative for each event to get the view that existed at its timestamp. Because our computation model uses a shared memory computing platform, STM is designed to address the shared state problem by maintaining multiple versions of each state variable.

Conventional memory can be viewed as a one-dimensional array of values, which can be addressed by specifying a *spatial* address. STM, on the other hand, is organized as a two-dimensional array (see Figure 2), in which each memory location is addressed by a row (*spatial*) as well as a column (*temporal*) coordinate. An event with timestamp T will perform reads and writes to the variable with T as the temporal coordinate. The state variables used by the simulator are mapped to a set of *data objects*, so the spatial coordinate is an object number. In general, each data object includes many state variables. A row of memory locations in STM can be viewed as a record of the history of object values across simulated time. When an event modifies a shared variable for the first time, STM creates a new copy, called a *version*, for that variable, rather than overwrite the previous contents. Any subsequent modifications to the variable by this event are made to the newly created version. By saving multiple versions of state variables, STM can provide multiple views of system state corresponding to different points in simulated time. A snapshot of the system state at any point of simulated time consists of the value of each state variable at that simulated time.

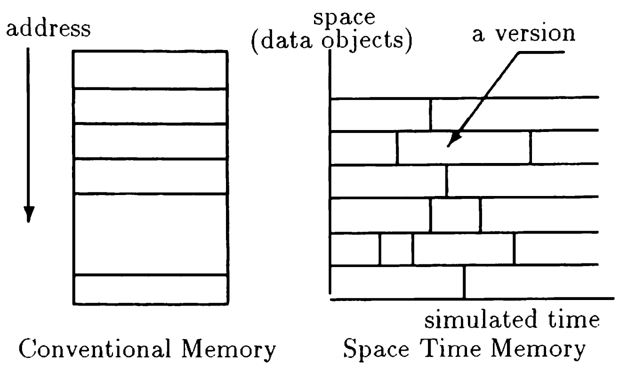

Figure 2: Memory address: conventional memory vs. space-time memory

STM provides the following operations which correspond to the four state primitives (i.e., **NS**, **RS**, **MS, and DS**) described earlier:

MakeObj(size): var_id allocates memory for data objects(s). It allocates a new block of memory containing **size** bytes, and returns a handle (**var_id**) for referencing this data object. It is an implementation of the **NS** primitive.

ReadObj(var_id): version reads the contents of a data object. It searches through the versions of the object (**var_id**), and returns a pointer to the most recent **version** with timestamp less than that of the event invoking this primitive. It is an implementation of the **RS** primitive.

WriteObj(var_id, value) modifies a data object. It creates a new version of the data object (**var_id**) whose new contents will be **value**. Events that have accessed versions of this object at higher points in simulated time than the event invoking this primitive will be rolled back and re-executed. It is an implementation of the **MS** primitive.

EraseObj(var_id) reclaims the memory allocated to a data object. It deallocates the memory which is allocated to a data object(**var_id**) by the operation **MakeObj**. To allow rollback, the implementation of this function must ensure that the actually deallocation of memory is not done until the deallocating event is committed. That is, when a data object is to be erased, it is indicated with a mark. If the erase operation is rolled back, the mark is cleared and the object un-erased. It is an implementation of the **DS** primitive.

Rolling back an event entails undoing any actions that modify the state or event set of the simulation, and may result in a cascade of rollbacks, as in Time Warp.

During a simulation, STM maintains the following information for each data object: (1) the simulated time at which it was created, (2) the simulated time at which the object was deleted, and (3) a list of versions. Each version also records: (1) the creation time, i.e., the timestamp of the event that executed a write to create this version, (2) the event that created the version, and (3) a list of events that have read this version. With this information, STM is able to determine the set of events that need to be rolled back on a write operation.

4.2 Event Set Operations

Optimistic parallel execution uses rollback to correct erroneous computations. Thus the system must be able to undo the operations on the event list. The event list can be either centralized or distributed. Rolling back the schedule operation (i.e., SE primitive) is accomplished by using an anti-message like mechanism (similar to Time Warp [Jefferson, 1985]) to annihilate the previously scheduled event(s) (details are described in [Fujimoto, 1989]).

To allow rollback, the retraction operation (i.e., DE primitive) must not discard the retracted event from the event list. Instead, it leaves the event in the event list and marks it as retracted. Rolling back a retraction operation need only remove the mark of the previously retracted event [Lomow *et al.*, 1991]. To enable rollback, each event carries the following information concerning the operations it performed: (1) events it scheduled, (2) events it retracted, (3) variables it created, (4) versions it created, (5) versions it read, and (6) variables it deleted. Discarding a version also implies that all of the events that read or wrote any newer version of the state variable must be rolled back.

In many simulation languages, an event may examine the event list (i.e., EE primitive). Because the operations may depend on the outcome of this examination, it is necessary to detect changes in the outcome caused by changes in the event list. One approach is to store a representation of the event list in the STM. When an event examines the event list, the information is recorded. Subsequent changes to the event list at earlier points in simulated time will necessitate a computation to determine if a rollback is necessary. This approach is under investigation.

5 CASE STUDY: SIMSCRIPT II.5

We choose SIMSCRIPT II.5 as the language for this study for two reasons. First, SIMSCRIPT II.5 is a widely used language, so automatic parallelization of SIMSCRIPT II.5 programs has some practical significance. Secondly, SIMSCRIPT II.5 is a relatively old programming language, and was not designed with any consideration for parallel processing. This makes it a challenging language for automatic parallelization of existing sequential DES languages. Details of the language are described in [CACI Products Company, 1989, Russell, 1989].

The structure of a SIMSCRIPT II.5 program consists of a declarative part and an executable part. The declarative part, called the PREAMBLE, defines all of the global variables, modeling elements such as events and processes, and metrics of the system to be measured. The executable part contains a mandatory MAIN routine, a routine for each type of event and process declared in PREAMBLE, and miscellaneous supporting routines for function and procedure calls.

SIMSCRIPT II.5 was designed to use a sequential execution mechanism. When an event/process is scheduled, a data structure (called event/process notice) representing the event/process is inserted into a global event list. The notices in the global event list are sorted by simulated time. One notice is removed from the global list at a time, and the corresponding event/process routine is executed. The difference between an event and a process is when an event finishes its execution, its event notice is discarded. When a process finishes its current execution, its notice is saved for rescheduling.

Specific SIMSCRIPT II.5 constructs and their implementation in the parallelization system are described below. Each construct starts with the description of its usage and operation in the sequential execution, followed by a brief discussion of how it is implemented in the parallelization system. In the parallelization system, a SIMSCRIPT process is represented by a state object that is scheduled and unscheduled (retracted) from the list of pending events to implement various behaviors, e.g., blocking or resuming execution of the process. State variable defined in the process are stored within the process's event notice, and are checkpointed to allow rollback.

CREATE, for creating an event or a process. On executing this statement, a notice is created (i.e., memory is allocated), but not inserted into the event list. For example, the statement, **CREATE A takeoff**, allocates memory for a notice representing a takeoff event/process. The parallel implementation of this construct allocates memory from STM (using **MakeObj**) for the event/process notice.

DESTROY, for destroying an event or a process notice. It is assumed that the event/process notice is not currently in the event list. Because the destroy may be rolled back, this operation is implemented by marking the notice as destroyed (using **EraseObj**), rather than immediately reclaiming the memory. The memory is freed only when the operation is committed.

SCHEDULE, for scheduling an event. For example, the statement, `SCHEDULE A takeoff NOW`, schedules an event called `takeoff` at the current simulated time. This construct can be directly mapped to the event scheduling primitive.

CANCEL, for canceling an event. For example, the statement, `CANCEL takeoff`, cancels the takeoff event. It can be directly mapped to the event retraction primitive. As described in section 4.2, the event is marked as cancelled, but still remains in the event list.

ACTIVATE/REACTIVATE, for activating a process, or reactivating a suspended process (see **SUSPEND** below). For example, the statement, `ACTIVATE A customer IN 10 MINUTES`, will start a process called `customer`. The starting time for this process is 10 minutes from the current simulated time. To implement this operation, a function called **ProcActivate** is defined that creates a process notice (see **CREATE**), and schedules an event for the process (see **SCHEDULE**).

WORK/WAIT, for a process to denote the passage of simulated time. For example, the statement `WAIT 10 MINUTES` puts the event notice for the currently executing process back into the event list with a timestamp set to the current simulation time plus 10 minutes. This statement signals the end of the current event. To implement this operation, a function **ProcWork/ProcWait** is defined in the parallelization system. This function uses the scheduling primitive to schedule an event and releases the processor to another event.

INTERRUPT, for interrupting a process. For example, the statement, `INTERRUPT customer`, temporarily removes the process notice denoting `customer` from the event list. The removed process notice must be kept intact for resumption. A process can be interrupted only if it is executing a **WORK/WAIT** statement. When interrupted, the amount of remaining waiting/working time is saved. The parallelization system defines a function called **ProcInterrupt** that uses the retraction primitive to retract the event for the interrupted process, then records the remaining waiting/working time.

RESUME, for resuming an interrupted process. For example, the statement `RESUME customer`, puts the `customer` notice back into the event list. The parallelization system defines a function called **ProcResume** that uses the scheduling primitive to schedule an event for the resumed process. The scheduled event has its timestamp set to that of the current time plus the saved remaining waiting/working time of the process when it was interrupted.

SUSPEND, for a process to suspend itself. This statement signals the end of the current execution. It keeps the notice intact. A parallel implementation causes the current event to release the processor to another event.

6 THE PARALLELIZATION SYSTEM

The parallelization system consists of two parts. The first is a compiler that takes a SIMSCRIPT II.5 program and produces an equivalent C program with embedded calls to Space Time Memory and event manipulation primitives. The second is a run-time library of routines for optimistic parallel execution, as described earlier.

A data structure called the *maptable* is defined to provide a mechanism for mapping simulation state variables into STM data objects. In the translated program, a maptable entry is defined for each state variable, and any access to it must go through the maptable. Each entry of the maptable has two fields: a pointer to a data object in STM, and an address offset (in bytes) of that state variable within the data object. Any access to a global variable must uses the functions defined by STM and typecast to the appropriate type. Similarly, event list operations and process interactions are translated into function calls to the run-time library.

The translated program (with embedded library calls) is then compiled and linked with the run-time library. This library provides a function for each SIMSCRIPT II.5 simulation construct discussed in the previous section.

7 PRELIMINARY RESULTS AND DISCUSSION

A simulator of the parallel execution has been developed for verifying the approach and determining the amount of parallelism available in the application. Later, we describe initial performance measurements of an operational parallel implementation executing on a Kendall Square Research KSR-1 multiprocessor. The current implementation of the simulator includes features of STM and interactions between processes.

7.1 The Simulator

The simulator uses a light-weight threads package called CTHREAD [Mukherjee, 1991]. A scheduler is used to dispatch scheduled events to processors. Scheduling is based on the smallest timestamp policy, i.e., the globally smallest timestamped event is executed next. Once an event is assigned to a processor, it does not release the processor unless it finishes its execution or is rolled back. Rollbacks are performed immediately once a data dependency violation is detected. An event attempting to read an STM version which is being written by another event must wait until the write operation is completed.

The execution time of a translated program is measured by counting the number of statements that are executed. We assume that each statement of the translated C program takes one unit of simulated time to execute. The generation of the embedded code to count the number of executed statements is automated. We present potential speedup results where the overhead of rollbacks and STM accesses in the parallel system and event list manipulation in the sequential simulator are negligible. Thus, these results optimistically assume that parallel synchronization overheads are comparable to priority queue overheads in the sequential simulator. Overheads in the parallel implementation include traversal of STM to locate a proper version of an object, logging access information, clearing premature accesses and execution (i.e., rollbacks), and maintaining the event list. These experiments give an indication of the amount of parallelism that is available in the application.

Three SIMSCRIPT II.5 benchmarks were used in our initial experiments. The first example is an event-oriented queuing network simulation. The second is a process-oriented sharks world [Conklin *et al.*, 1990] simulation, and the third is a traffic control simulation. The performance metric used is the potential speedup (P_SU), defined as the ratio of execution time of the parallel implementation and the execution time on a single processor (again, with overheads in both assumed to be zero).

7.2 Example 1: 64 Node Hypercube Queuing Network

The first example is a simulation of a closed queuing network configured as a 64-node hypercube. This benchmark bears some resemblance to simulations of communication networks, and has been a widely used benchmark in PDES research. In this example, messages may be sent from one node to any adjacent node. This benchmark is programmed with only events (i.e., no processes are used). Two events are used to model each hop. One is for the departure, and the other for the arrival at the adjacent node. Messages at a node are processed in first-come-first-serve order. An arrival event schedules a departure event with timestamp increment corresponding to the waiting time in the queue plus a service time. The service time of each arrival is selected from an exponential distribution with a mean of 5 time units. The simulated system is a closed network with zero-delay transmission, so a departure event schedules an arrival event with zero timestamp increment. Thus, the application is *not* written to exploit "lookahead." There is a server at each node. Each node uses two variables: the local clock, and the number of messages at the node. Both are mapped to STM data objects.

Several runs are performed with message populations ranging from 64 to 256. When the parallelized version runs on the simulator, the measured potential speedup is significant. With 16 processors, the average potential speedup is about 15. The ratio of potential speedup and the number of processors decreases slightly as the number of processors increase because of interference between events at the same node, causing more rollbacks. However, even with 64 processors, the potential speedup could still be as high as 46.55 when the message population is 256 (see Figure 3). As shown in Figure 4, the percentage of all events executed that are committed (the efficiency) decreases from 95.6% to 64.5% as the number of processors increases from 16 to 64 when the message population is 256. The decrease in the committed event rate is more pronounced when the population is small.

7.3 Example 2: Shark's World

This process-oriented simulation models the activities in an ocean containing two types of creatures: fish and sharks [Conklin *et al.*, 1990]. The ocean is divided into a two-dimensional matrix of sectors with toroidal wrap around connections at the edges. Sharks swim faster than fish, and can attack fish, but

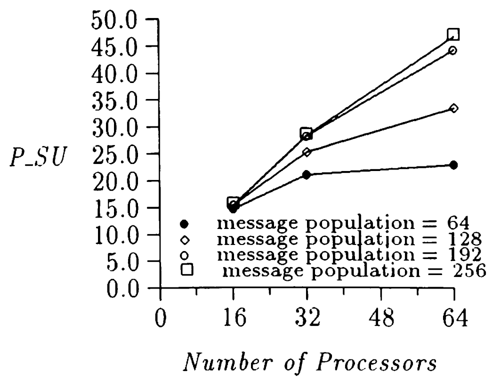

Figure 3: Queuing network: potential speedup (P_SU) vs. number of processors

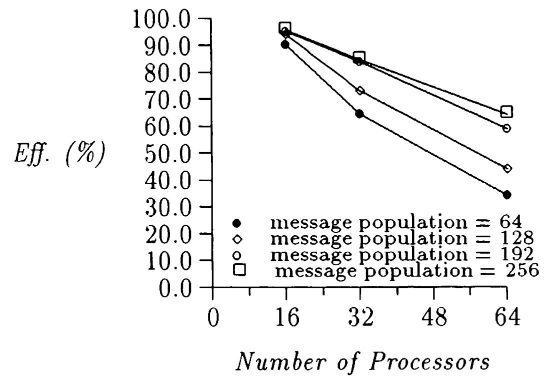

Figure 4: Queuing network: efficiency vs. number of processors

not each other. The benchmark was designed to capture behaviors found in combat simulation models.

In this simulator, each creature is modeled as a process, and the location and status (alive, or dead) of each creature as shared variables. A creature can swim from one sector to any adjacent sector. When a shark enters a sector, it will attack and kill all the fish in that sector before it leaves. The speed and direction are fixed for each creature. The speeds of sharks are uniformly distributed between 50 and 100 units per second, and those of fish between 10 and 20. The direction of each creature is uniformly distributed between 0 and 360 degrees.

The size of the ocean is 3200 by 1600 units, and is divided into a 8×4 matrix. The initial ratio of fish to sharks is 6 to 1. The number of resident creatures varies in each run, ranging from 56 to 224 creatures. When the parallelized sharks world program was run on the simulator, the potential speedup for 8 processors is as high as 7.45. However, for 32 processors, the potential speedup is at most 16.93 with 224 creatures (see Figure 5). This is because interactions between creatures will limit the amount of concurrent execution. For example, when more processors are used, more creature processes can be executed concurrently. Since the sharks in the same sector may attack the same fish, therefore, more interactions are observed.

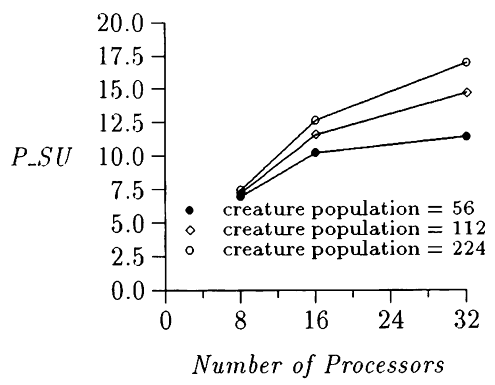

Figure 5: Sharks world: potential speedup(P_SU) vs. number of processors

7.4 Example 3: Traffic Control Simulation

The traffic control simulation consists of a number of intersections which are controlled by traffic lights. To reach its destination, each car has to drive from its starting point through a sequence of intersections. When arriving at an intersection, a car will be enqueued in the lane if other cars are present. It is dequeued when it is its turn to cross the intersection. Three types of processes are used, a car process, which models the itinerary of a car, a light control process, which changes the status of all lights, and a generation process, which continuously generates a car process. In this benchmark, traffic lights are modeled as shared states because each car must read their status (i.e., color).

Two experiments with different problem sizes were performed. The first has a street map consisting of 3×3 intersections, and the second 9×9 intersections. As can be seen (Figure 6), the potential speedups in both cases are very poor. This is because the simulation program was written with a centralized light control process, that sequentializes the execution by changing the status of every traffic light. Therefore, the parallelism in the original application was only slightly more than 2 – derived from the independence between the car generation process, and the light control process.

A restructuring of the traffic benchmark was performed to improve the parallelism. Rather than using a single process to control the lights at *all* intersections, a separate process could be used for *each* intersection. With this transformation, not every car

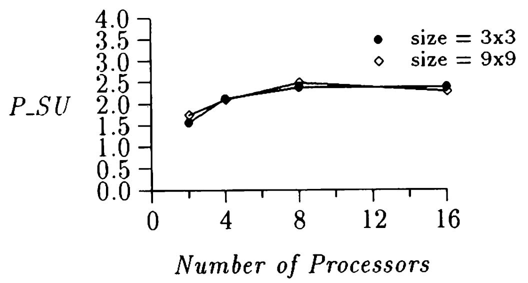

Figure 6: Traffic control: potential speedup(P_SU) vs. number of processors

process has to interact with the same light control process as before. Therefore, the parallelism in the program is proportional to the number of intersections (i.e., the number of light control processes). The potential speedup is shown in Figure 7.

Although the above transformation was performed manually in our experiments, it could be automated. The optimization is similar to that commonly used in parallelizing compilers for concurrent execution of DO-loops.

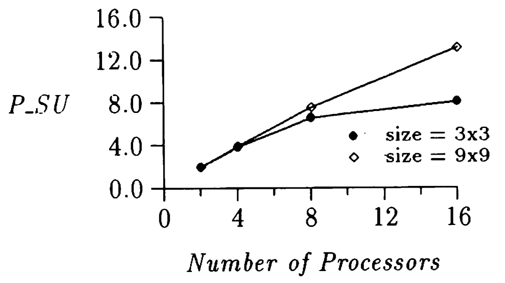

Figure 7: Traffic control (modified): potential speedup(P_SU) vs. number of processors

7.5 Experiments on the KSR

To evaluate the performance that can be actually realized in practice, an initial implementation of the automatic parallelization system for SIMSCRIPT II.5 programs has been developed. This system is now operational on a Kendall Square Research KSR-1 multiprocessor, enabling evaluation of the affect of overheads on performance.

Figure 8 shows the measured speedup as the number of KSR processors is varied for the hypercube simulation described earlier. Speedup is measured relative to execution on a single KSR processor. The message population is set at 256. To evaluate the effect of overheads, speedup is shown for the original parallelized program as well as the same program with the computation granularity of each event increased using a spin loop by 3 and 25 milliseconds. For the original program, speedup reaches 3.5 with 8 processors, but additional processors provide no additional performance benefit; in fact, speedup declines somewhat because the number of rollbacks increases significantly, incurring additional overheads. For this case, the measured efficiency decreases from 63% (8 PEs) to 39% (16 PEs). As the computation granularity of the simulations is increased, performance improves. With 25 milliseconds added to each event, approximately 70% of the potential speedup reported in figure 3 is obtained. A challenge for the future is to enable better performance for small granularity event computations.

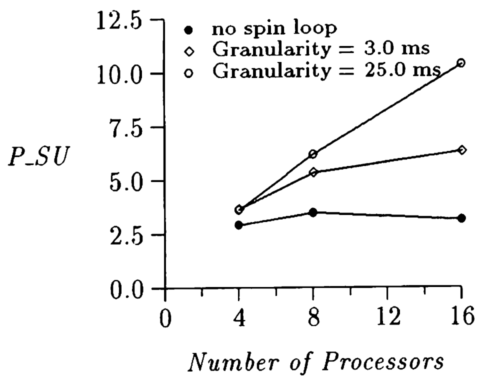

Figure 8: Queuing network: speedup (SU) vs. number of processors on KSR

8 CONCLUSION

In this paper, a system is proposed to adapt sequential discrete event simulation programs to a parallel platform based on optimistic execution. A standard framework including a set of simulation primitives is defined. We then show how these primitives can be parallelized with optimistic synchronization.

A prototype simulator was developed to implement the proposed parallelizing system. The simulator includes a compiler that transforms SIMSCRIPT II.5 programs into equivalent C code with calls to a library that provide run-time support for optimistic parallel execution. The system was demonstrated on a SKR-1 multiprocessor to obtain good speedup for at least one benchmark when computation granularity is high (tens of milliseconds).

The potential speedup data derived from our experiments show that for several applications, sufficient parallelism is available to offer significant speedups. Program transformations may be required to exploit parallelism in certain circumstances. We observed that techniques developed for parallelizing compilers to unfold DO-loops can provide significant benefit in some situations.

It is clear that much research remains before automatic parallelization of DES programs can be considered a viable technology. The extent that the parallelization process can be fully automated is unclear. However, it is equally clear that new techniques to aid the transition to utilization of parallel discrete event simulation techniques are sorely lacking, and automatic and semi-automatic parallelization techniques offer some hope in fulfilling this need.

ACKNOWLEDGEMENTS

This research was support by NSF grant number CCR-8902362. Technical assistance from CACI Products Co., and Ana Marjanski in particular, is gratefully acknowledged.

REFERENCES

CACI Products Company. 1989. *SIMSCRIPT II.5 Reference Handbook.*

Comfort, J. C. 1991. Environment Partitioned Distributed Simulation of Queueing Systems. In *1991 Winter Simulation Conference Proceedings*, pages 584–592.

Conklin, D., J. Cleary, and B. Unger. 1990. The Sharks World (A Study in Distributed Simulation Design). *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):157–160.

Fujimoto, R. M. 1989. The Virtual Time Machine. *International Symposium on Parallel Algorithms and Architectures*, pages 199–208, June.

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

Ghosh, K. and R. M. Fujimoto. 1991. Parallel discrete event simulation using space-time memory. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 3, pages 201–208, August.

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

Kay, I. M. 1972. An Over-the-shoulder Look at Discrete Simulation Languages. In *AFIPS Conference Proceedings 40*, pages 791–798.

Kiviat, P. J. 1967. Development of Discrete Digital Simulation Languages. *Simulation*, pages 65–70, February.

Lomow, G., S. R. Das, and R. M. Fujimoto. 1991 Mechanisms for User Invoked Retraction of Events in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 1(3):219–243, July.

Mukherjee, B. 1991. A Portable and Reconfigurable Threads Package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June.

Paterra, F., C. M. Overstreet, and K. Maly. 1990. Distributed Simulation: No Special Tools Required. In *Proceedings of 1990 Winter Simulation Conference*, pages 423–427.

Polychronopoulos, C. D. 1988. *Parallel Programming and Compilers.* Kluwer Academic Publishers.

Pritsker, A. B. 1974. *The GASP IV Simulation Language.* John Wiley & Sons.

Russell, E. C. 1989. *Building Simulation Models with SIMSCRIPT II.5.* CACI Products Company.

Spriet, J. A. and G. C. Vansteenkiste. 1982. *Computer-Aided Modelling and Simulation.* Academic Press.

Szymanski, B. K., editor. 1991. *Parallel Functional Languages and Compilers.* ACM Press.

Wolfe, M. 1988. Multiprocessor Synchronization for Concurrent Loops. *IEEE Software*, January.

Zeigler, B. P. 1976. *Theory of Modeling and Simulation.* Wiley-Interscience Publication.

AUTHOR BIOGRAPHIES

JYA-JANG TSAI is a PhD student at College of Computing, Georgia Institute of Technology. His research interests include automatic parallelization of discrete event simulations and performance evaluation.

RICHARD M. FUJIMOTO is an Associate Professor in the College of Computing at the Georgia Institute of Technology. He is an area editor for the ACM's *Transactions on Modeling and Computer Simulation.* His current research interests include computer architecture and parallel simulation.