*Proceedings of the 1998 Winter Simulation Conference*
*D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.*

# EFFICIENT PROCESS INTERACTION WITH THREADS IN PARALLEL DISCRETE EVENT SIMULATION

Reuben Pasquini
Vernon Rego

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, U.S.A.

## ABSTRACT

Parallel discrete event simulation (PDES) decreases a simulation's runtime by splitting the simulation's work between multiple processors. Many users avoid PDES because it is difficult to specify a large and complicated model using existing PDES tools. In this paper we describe how the ParaSol PDES system uses migrating user level threads to support the process interaction world view. The process interaction world view is popular in sequential simulation languages and is a major departure form the logical process view supported by most PDES systems.

## 1 INTRODUCTION

Discrete event simulation (DES) is a technique that exploits a computer to model a system whose state changes (stochastically) at discrete points in time. A simulation program operates on a model's *state* variables during each of a sequence of time-ordered *events*. It is not enough for a simulator to process events quickly; a simulation language must also export an API with which large models may be simply specified and modified. Parallel discrete event simulation (PDES) algorithms attempt to speed up the execution of a DES by distributing simulation workload across distinct processors. We believe that PDES offers great promise for meeting the simulation needs of developers of increasingly complex systems.

A simulation language presents to a developer a *world view*. A world view is simply the programming interface supplied by a language with which a developer must describe his simulation model. Three world views which different simulation languages (both sequential and parallel) support are the *event*, *active server*, and *active process* world views (Carson 1993).

The event-scheduling world view allows a simulation developer to describe his model in terms of a set of simulation events which act upon simulation objects. For example, a simulation of a grocery store check-out line might define *request*, *service*, and *complete* events which act upon a check-out object. A *request* event signifies the arrival of a customer requesting service. The *request* event $e_r$ is placed in a queue if the server is busy; $e_r$ eventually generates a *service* event when the customer gets served. Finally, the *complete* event signifies that the customer has been served.

The active-server world view allows the developer to express his simulation in terms of active server entities which exchange typed messages between each other. The active-server world view and event-scheduling world view are in some sense duals of each other. In the event-scheduling world view actions are associated with events which manipulate data stored in servers. In the active-server world view actions are associated with servers which manipulate local data and data carried in messages (which can be viewed as events). For example, in the check-out line example described above, we would write a server procedure to describe the actions of the check-out clerk. The clerk routine waits for the arrival of a message (customer), then executes some service routine for the customer message. Finally, the clerk generates a new message to pass the customer on to another object.

The active-process world view is the most popular programming interface for specifying many types of models. In the active-process world view a developer describes his model in terms of the actions of active processes on simulation objects. For example, in the check-out line model described above the simulation would be expressed in terms of the *customer's* actions upon the check-out counter.

This paper outlines the difficulties in implementing the active-process world view in parallel discrete event simulation, and describes how the ParaSol PDES system addresses these challenges. Section 2 introduces several

PDES systems, and section 3 presents the PARASOL system. We describe how to implement an efficient user level threads system for PDES in section 4. We go on to outline the unique demands which PDES places on our thread system and how we satisfy these demands in section 5. We present the results of simple experiments which we conducted to evaluate the performance of our thread system design in section 6. Finally, we conclude in section 7.

## 2 RELATED WORK

### 2.1 Parallel Simulation Concepts

PDES algorithms attempt to speedup the execution of a DES program by distributing the simulation workload across multiple processors. A DES executes a time-ordered sequence of simulation *events*. Each event may access one or more simulation objects and schedule one or more future events. The state of the simulated system is defined by the state of all simulation objects. The order in which events execute is determined by a *virtual time* which is defined by event time–stamps. Events execute in nondecreasing time–stamp order so that *virtual time* always advances.

It is natural to think of parallelizing DES programs by distributing all the simulation events across multiple processors. Given $n$ processors and $m$ events, each processor would ideally handle $m/n$ events, suggesting an ideal speedup of $n$. Unfortunately, distributed events typically don't access simulation objects in time–stamp order. For example, processor $p_1$ may execute an event $e_1$ with time–stamp $t_1 = 1$ after processor $p_2$ executes an event $e_2$ with time–stamp $t_2 = 2$. If $e_2$ happens to access a shared simulation object (i.e., an object shared by $p_1$ and $p_2$) before $e_1$ is able to access the object (say, because of processor or network delay), then the parallel execution witnesses $e_1$ and $e_2$ access the shared object in an order that is different from the order in which these events access the object in a sequential execution ($e_1$ followed by $e_2$).

A PDES must employ an algorithm which ensures that events execute in a *causally consistent* way. A simulation is causally consistent if each simulation object is accessed by events in nondecreasing time–stamp order. The time warp algorithm (Jefferson 1985) is an example of an *optimistic* algorithm for PDES. It is optimistic in the sense that each processor executes every event it knows about in time–stamp order under the optimistic assumption that causality is not being violated. At any point, however, a processor may receive an event (from another processor) whose time–stamp indicates that it should already have been processed; such an event is called a *straggler*. After detecting a straggler, a processor *rolls back* to a system state that corresponds to a time–stamp which is less than the straggler's time–stamp. Execution continues from this point, and the straggler is processed in the right time–stamp order. A successful optimistic PDES system must minimize the runtime costs of *state-saving* (for potential rollback), *rollback* (to recover state), *global virtual time (gvt)* computation (to determine the actual simulation time) and *interprocessor communication*, in order to deliver speedup relative to a sequential simulation.

### 2.2 PDES Systems

A traditional PDES system expresses a model in terms of communicating logical processes. The PDES system maps each logical process (LP) to a node of a multiprocessor, and uses interprocessor communication to allow LP's on different processors to communicate with each other.

Several LP based PDES systems have been implemented. The *Georgia Tech Time Warp* (*GTW*) PDES library simulates models described with the event-scheduling world view on shared memory parallel computers (Fujimoto 1989). The *WARPED* (Martin and McBrayer 1997) PDES system also uses the event-scheduling world view. *Maisie* is a parallel simulation language that supports the active-server world view. *Maisie* improves upon *GTW* by making constructs for parallel execution more transparent to the user (Bagrodia 1991). The *SIMKIT* language is another system that supports an active-server world view (Gomes et al. 1995).

To the best of our knowledge, PARASOL and *APOSTLE* are the only two PDES systems which support the active-process world view. The *APOSTLE* system manages simulation process state as continuations constructed with compile time transformation of simulation code. *APOSTLE* employs the semi-conservative breathing time-buckets algorithm to enforce the causality constraint on shared memory architectures (Booth and Bruce 1997). PARASOL implements simulation processes as user level threads. PARASOL uses the optimistic time warp synchronization algorithm on distributed memory architectures.

## 3 PARASOL BACKGROUND

PARASOL is a process- and object-oriented parallel simulation language developed for distributed-memory multiprocessors and workstation clusters (Mascarenhas, Knop, and Rego 1995). PARASOL's user interface is an object-oriented derivative of the user interface for the popular sequential simulation language *CSIM* (Schwetman 1986). PARASOL's sequential runtime performance is competitive with *CSIM's* performance on equivalent models; this makes it easy to relate PARASOL's performance with the performance of commercial sequential simulators.

PARASOL has a simple layered design (see figure 1). The bottom *system* layer of the PARASOL architecture is shared by a threads system and a message passing system. PARASOL's *kernel* layer manages the time warp mechanism and exports basic simulation primitives to the *domain* and *application* layers. The *domain* layer implements domain specific simulation objects whose state can be transparently saved and restored by the simulation kernel. Finally, the user application defines the *application* layer. Application code should not have to take into account special mechanisms needed for parallel simulation.

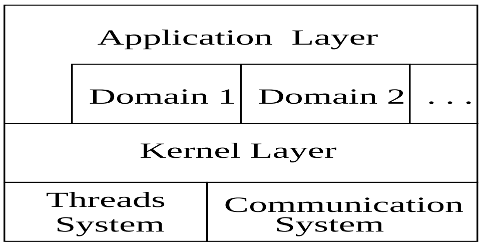


Figure 1: PARASOL's Software Architecture

## 4 THREADS AS SIMULATION PROCESSES

PARASOL implements each simulation process as a thread. Since a simulation may involve thousands of concurrently executing threads, PARASOL requires a fast and efficient thread system to achieve good runtime performance. Although user level threads take less time to create and context switch than operating system kernel threads (Kleiman, Shah, and Smalders 1996), user thread manipulation is still expensive when compared to event handling in simulation systems supporting the event-scheduling world view. We have developed a high performance user level thread system for managing simulation process state in parallel and sequential discrete event simulation systems. This thread system is specially designed to support the unique demands (checkpointing, rollback, thread migration) of parallel simulation systems in a way that significantly reduces the cost of using threads to implement simulation processes.

### 4.1 Thread Library Implementation

Let's consider the implementation of a user level thread library for the C programming language. In this system, when a thread $h_a$ yields control of the processor, a thread scheduler routine selects the next thread $h_b$ to run. Thread $h_b$ keeps control of the processor until $h_b$ either yields control back to the scheduler or $h_b$ completes execution. We call this threads system the *SAM* system. Although SAM is very simple, it's a good starting point for designing more complicated systems.

SAM has a simple two routine API.

- `void th_create( void (*f)() )` – create a new thread context
- `void th_yield()` – yield control to the scheduler

When a running thread $h_a$ calls `th_create()`, the scheduler creates a thread context object for a new thread $h_b$ and places $h_b$'s context onto a scheduling queue. In a simulation system, the simulation calendar is responsible for scheduling simulation processes in time-stamp order. Therefore, the thread scheduling queue is actually the simulation calendar in a thread based simulation system.

When thread $h_a$ yields control of the processor (by calling `th_yield()`), the scheduler places $h_a$'s context onto the the scheduling queue, removes $h_b$ from the front of the queue, and transfers control of the processor to $h_b$. In a simulation system, a thread (simulation process) yields whenever the thread executes a `hold` operation or suspends itself (to be placed on a server's wait queue for example).

SAM uses the C-language `setjmp` and `longjmp` functions to context switch between user level threads. The `setjmp` function saves the transient executing environment of the currently executing thread into a *jump buffer* $j$. The scheduler can later resume $h_a$'s execution by calling the `longjmp` function to restore the environment saved in $j$. In order to force the processor to execute a new thread $h_c$ on a new stack, the scheduler constructs an "artificial" jump buffer $j$. The scheduler sets the jump buffer's environment variables so that when it calls `longjmp` on $j$, the processor is tricked into starting $h_c$'s execution on a new stack. This simple `setjmp` and `longjmp` context switching scheme works well on some architectures (including the PC), but others (like the SPARC) require an assembly code context switch routine.

### 4.2 Threaded Applications

The two most important overheads in SAM are context switching and stack management. The best method for minimizing these overheads depends upon the the nature of the application. We consider two types of threaded applications – applications employing few (less than 100) threads and applications employing many (100 or more) threads.

A simple application which uses few threads is the network *talk* application. This is a simple tool which allows two persons to communicate in (near) real time over the Internet through a keyboard–screen interface. We can imagine a simple two thread *talk* implementation. One thread monitors the network, and prints received messages to the screen. The other thread monitors the keyboard, and sends typed keys over the network.

Since this *talk* application has only two threads, SAM can optimize context switching time by allocating a large

(100 kilobyte to 1 megabyte) stack for each thread. During a context switch from thread $h_a$ to thread $h_b$, the scheduler simply flushes the processor's register state onto thread $h_a$'s stack (with `setjmp`), and moves the stack pointer to thread $h_b$'s stack (with `longjmp`).

An important overhead of allocating a personal stack to each thread is the large amount of memory each thread's stack requires. A large stack allows a thread to allocate objects on its stack, and have a deep function call path. Unfortunately, if a thread has a small execution footprint, then most of the stack is wasted.

### 4.3 Supporting Applications with Many Threads

Discrete event simulations are applications which can require thousands of threads. For example, a parallel discrete event simulation of a large ATM network may allocate a thread to route each simulated packet. This simulation involves several thousand concurrently executing threads. SAM can not afford to allocate a large stack to each thread in such an application.

SAM can support thousands of threads if it conserves memory by allowing multiple threads to share a single stack. Many threads can share a single stack if SAM's context switch routine copies thread state on and off of the shared stack at context switch time. For example, when a thread $h_a$ yields control of the processor to thread $h_b$, SAM copies $h_a$'s state off of the shared stack to a buffer for safe keeping. Next, SAM copies $h_b$'s state onto the shared stack, and resumes $h_b$'s execution (see figure 2).

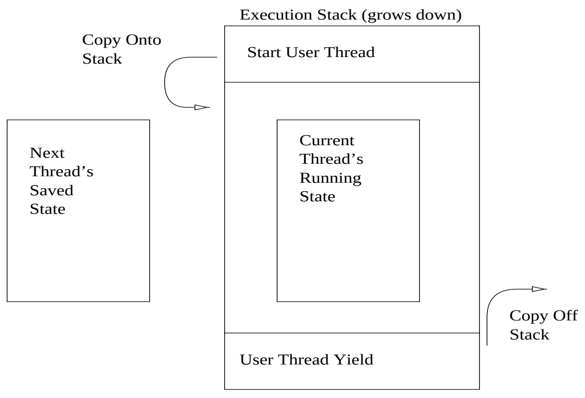

Execution Stack (grows down)
Copy Onto Stack
Start User Thread
Next Thread's Saved State
Current Thread's Running State
Copy Off Stack
User Thread Yield

Figure 2: Common Stack Memory Layout

Sharing a single stack between multiple threads saves memory by eliminating the memory wasted in the portion of a stack allocated to a thread but not used by the thread. For example, during a thread's lifetime its state may grow to a maximum size of 500 KB for brief periods of time when making calls to specific procedures. However, at context switch time (when the thread yields control of the processor), the thread may only be using 10 KB of the stack on average. If this thread has its own stack, then SAM must allocate (at least) 500 KB of memory for the thread's stack. Suppose a simulation requires 1024 of these threads to exist in the system, then SAM must dedicate (approximately) 500 MB of memory to thread stacks alone. However, if the 1024 threads share a single stack, then SAM only allocates 500 KB of memory for the single stack plus $1024 * 10$ KB $= 10$ MB of memory for buffers.

We can see that the stack sharing scheme is especially useful in applications with threads which periodically grow to a large size, but usually remain small. With the shared stack scheme SAM can allocate a buffer at context switch time just large enough to save the running thread's state. This eliminates the memory fragmentation present in the large portion of the stack not used by the thread.

## 5 THREADS AND DISTRIBUTED SIMULATION

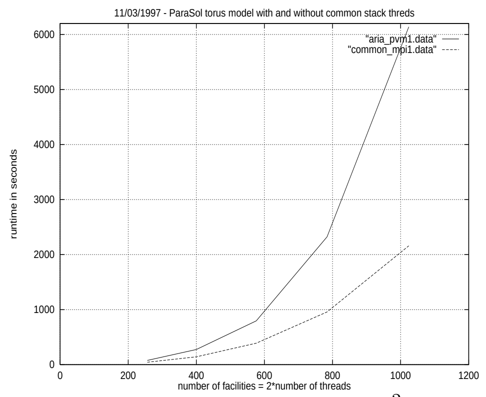
ParaSol places two unique demands upon its thread system to implement an optimistic simulation algorithm. First, SAM must be able to checkpoint and rollback a thread's state. Second, SAM must have the ability to migrate a thread from one processor to another processor.

### 5.1 Checkpointing Thread State

ParaSol implements an optimistic PDES algorithm which requires each processor participating in a parallel simulation to periodically checkpoint its local state. A processor checkpoints its local state by copying the read-write state of simulation objects and the stack of simulation threads to backup buffers.

In section 4.3 we describe how sharing a common stack between multiple threads reduces memory requirements in applications using many threads. When a running thread $h_a$ yields processor control to another thread, SAM copies $h_a$'s state off of the shared stack to a temporary buffer $b_a$. To checkpoint thread $h_a$'s state, SAM copies $b_a$ to a checkpoint buffer $c_a$. Before the next time $h_a$ runs, SAM copies $h_a$'s state from $b_a$ onto the shared stack and frees $b_a$ back to the memory pool.

The thread checkpointing algorithm described above is very inefficient. SAM can avoid copying $b_a$ to $c_a$ at checkpoint time by simply incrementing a reference count to $b_a$ and using $b_a$ as a checkpoint. This simple trick virtually eliminates the cost of checkpointing threads in ParaSol. Figure 3 compares the runtime of a ParaSol simulation whose threads share a single stack with the runtime of a simulation whose threads each have a separate stack.



**x-axis: objects in torus $= N^2$**
**y-axis: runtime in seconds**

Figure 3: PARASOL's Runtimes for the Torus on a Sparc 20 with State Saving Using a Shared Stack (dashed line) and Individual Stacks (solid line)

## 5.2 Thread Migration

PARASOL uses the SPMD (single program, multiple data) model to run parallel simulations over distributed memory multiprocessors. Each processor executes the same program code (executable), but each processor follows a different path of execution through that code as the simulation progresses. PARASOL implements a simple distributed shared memory system based on thread migration to manage interprocessor communication in a way that is hidden from the simulation user.

We must understand the structure of a PARASOL simulation before we can understand how PARASOL uses thread migration. A PARASOL simulation consists of simulation processes (threads) which access simulation objects. PARASOL statically assigns each simulation object $x$ to a processor at the beginning of a simulation. Each processor which does not host an object $x$ allocates a proxy object for $x$. When a simulation thread $h_a$ executing on processor $p_1$ attempts to access an object $x$ located on another processor $p_2$, $h_a$ accesses $x$'s local proxy on processor $p_1$. The proxy object executes code which forces $h_a$ to migrate to processor $p_2$ where $h_a$ can access $x$ directly (see figure 4).

SAM migrates a thread $h_a$ from processor $p_1$ to processor $p_2$ by passing $h_a$'s stack as a network message between the two processors. At the destination processor $p_2$, SAM must place $h_a$'s stack at the same memory address that the stack resided at on $p_1$. If SAM does not place the stack at the same address, then SAM must adjust the thread's recorded stack pointer by an offset corresponding to the difference between the stack's address at the source processor and the address at the destination processor (Mascarenhas and Rego 1996). Furthermore, $h_a$ may not

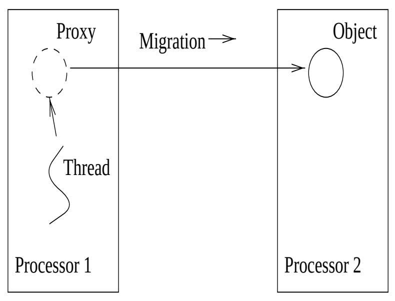


Figure 4: Thread Migration in PARASOL

use pointers which refer to objects on its stack since SAM can not update these pointers after migration without user intervention.

SAM can avoid the problems with stack address alignment by forcing each processor to allocate its thread stack at the same memory address. Therefore, threads execute on a stack located at the same address on every processor (see section 4). Migration safeness is one more benefit of using a shared stack in parallel simulation.

## 5.3 Accessing Objects After Migration

PARASOL implements a restricted distributed shared memory to allow simulation threads to access simulation objects which are distributed between processors. Each processor maintains a global block of memory $S$ at an address reserved for simulation objects. At the beginning of a simulation every processor executes a user-written routine which maps each simulation objects to a particular processor. Suppose that a simulation maps a `Storage` type object to processor $p_1$ by passing $p_1$'s pid to the object's constructor. When processor $p_1$ executes this command, $p_1$ allocates an object of type `Storage` at the next address $a_1$ available on $S$. Every other processor allocates a proxy object at address $a_1$ (see figure 5).

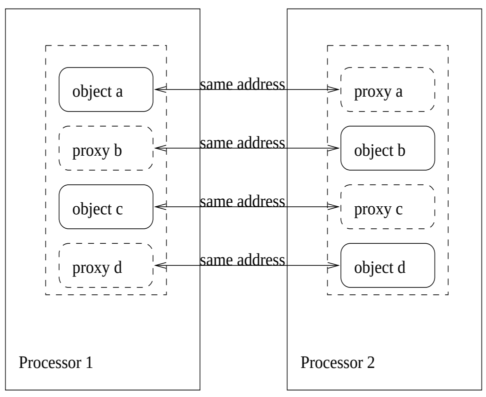


Figure 5: Distributed Shared Memory in PARASOL

A simulation object and its proxy are both instantiations of the same class. A proxy object discovers it's a proxy by comparing the id of its host processor with the id of the processor to which the object has been mapped. The id of an object's processor is passed to the object's constructor.

This simple distributed shared memory mechanism places several restrictions on the way a model may use the system's memory resources. First, a model must allocate every simulation object at the beginning of its simulation. Each simulation object (or its proxy) is visible to every simulation thread on every processor. Dynamic creation of simulation objects would require synchronization between every processor participating in the simulation. We chose not to support this expensive operation in ParaSol.

The second restriction on memory use is that every object (not a simulation object) dynamically allocated by a thread must be allocated on the thread's stack. This restriction is necessary because ParaSol's thread migration mechanism can not transparently migrate objects not located on the thread's stack. For example, suppose that a thread $h_a$ executing on processor $p_1$ allocates a string on the heap of processor $p_1$. When $h_a$ later migrates to processor $p_2$ the heap allocated string is left behind on processor $p_1$.

Thread objects are the one exception to the rule that a model can not allocate objects on the heap. ParaSol's object oriented thread interface is similar to the thread interface supported by the *Java* programming language. A model creates and destroys threads (simulation processes) throughout the life of a simulation. A model creates a new thread $h_a$ by allocating an object from a class which is a subtype of ParaSol's `Thread` class. The first time $h_a$ executes, the simulation kernel calls the `run` method of the object associated with $h_a$. During $h_a$'s execution, $h_a$ may safely access data stored in its thread object.

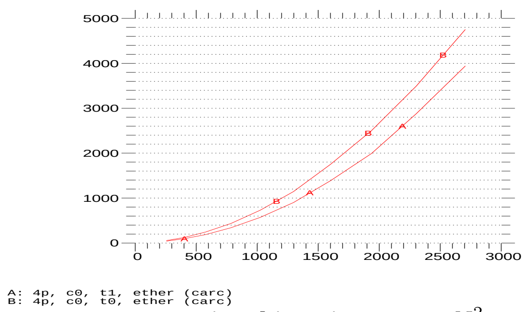
Other threads may not access $h_a$'s thread object since the object's memory address changes over $h_a$'s lifetime. When $h_a$ executes, ParaSol's kernel maps $h_a$'s object state to a reserved memory area located at the same address on every processor. This is the same trick we use to execute migrating threads on different processors by mapping the thread's stack to a well known memory address. Another restriction on a thread object is that the object may not contain pointers to heap allocated objects (other than simulation objects). Such pointers would be left dangling after the thread migrates.

## 6 THREADS AND SIMULATION PERFORMANCE

Before accepting threads as a basis for simulation processes we should compare thread runtime performance with the performance of lighter (but less user-friendly) implementation of simulation processes. We can carry out this kind of a comparison by allowing ParaSol to implement simulation processes with either threads or lighter "agents".

An agent is a C++ object which exports a `run` method (like a thread). The simulation kernel invokes the agent's `run` method each time the agent executes. When the agent suspends its execution (to hold on a server for example), then the agent's `run` method *returns* (the agent's stack is not saved as a thread's would be). The main difference between an agent and a thread is that an agent can not automatically save its execution state after suspending. This makes the code for a model written with agents more complicated than code written with threads.

The graph in figure 6 shows simulation run times for ParaSol's execution of a torus model using either agents or threads as simulation processes. This model is simply a two dimensional torus of $N \times N$ servers over which $N \times N/2$ simulation processes randomly migrate. We can see that a simulation which implements simulation processes with agents runs faster than a simulation of the same size model which implements simulation processes with threads. We expected this result since an agent does not collect an expensive checkpoint of its stack state after each execution. A thread must checkpoint its stack state after each execution to allow another thread to run on the shared stack.



**x-axis: objects in torus** $= N^2$
**y-axis: runtime in seconds**

Figure 6: Runtimes Using Threads (line $B$) and Agents (line $A$) on 4 Sun Sparc5 Workstations with 10 Mb/s Ethernet

Considering our thread system's stack copying overhead, it is surprising that the thread based system performs so well compared to the agent based system. In fact, we can see that for the longest running simulation we sampled in figure 6, the thread based system had a runtime only 1.20 times the runtime of the agent based system. Is this a reasonable result? We can answer this question by measuring the context switch cost of our agent and thread based systems in isolation. We can measure the context switch overhead with the following simple test. We simulate a model which involves only two simulation processes. When each simulation process executes, it simply suspends itself and yields to the other simulation process (forcing a context switch). The simulation continues until 100000 context switches complete, then the simulation exits. We compare the run times of the thread and agent based systems on this *switching* model to determine the relative cost of context switching between agents compared to switching between threads.

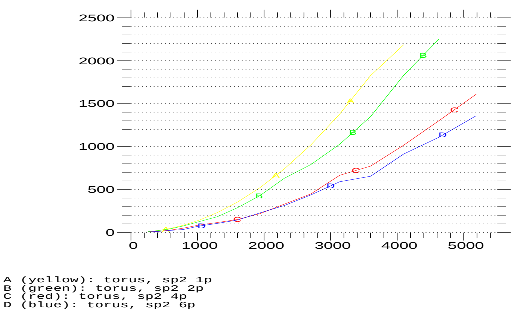
The results of our test are shown in table 1. These measurements tell us that a thread context switch takes 1.67 times the time of an agent context switch. The larger 1.67 ratio ($R1$) between thread and agent context switch times seems more reasonable than the 1.20 ratio ($R2$) between thread and agent simulation times (see figure 6) in light of the stack copying overhead associated with our thread system. Ratios $R1$ and $R2$ are different because they compare measurements of different quantities. Ratio $R1$ compares the context switch run times of thread and agent systems. Ratio $R2$ compares the run times of actual simulations which perform many operations (such as calendar management and event execution) in addition to the context switching.

In other words, although threads may context switch in 1.67 times the time that agents switch in, a thread based simulation may finish in 1.20 times the time of an agent based simulation. This is possible since a simulation involves other overheads in addition to context switching between simulation processes. Unfortunately it is difficult to predict the runtime costs of using threads in any particular simulation since thread switching overheads are application dependent. For example, it is more expensive to context switch between threads with large runtime stacks than between more lightweight threads. On the other hand, if each thread completes a lot of computational work before suspending (high granularity), then the importance of context switching overhead to the overall runtime of a simulation decreases. Our experiments involve threads with shallow runtime stacks and low computation granularity.

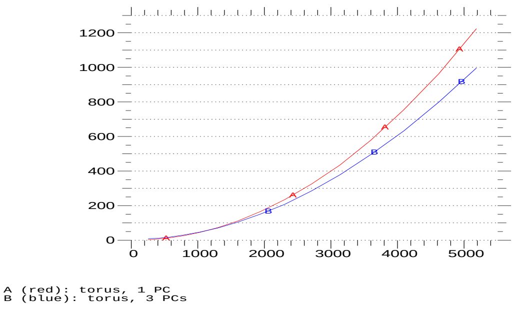
The final measure of a PDES system's performance is the amount of speedup the system yields on common benchmarks. Figure 7 shows that ParaSol's runtime on the torus benchmark decreases as the number of processors employed in the simulation increases.

Table 1: Time for 100000 Context Switches with Agents and Threads

| | mean | variance | samples |
|---|---|---|---|
| threads | 29.35 | 12.82 | 100 |
| agents | 17.58 | 1.14 | 100 |



(a) Scalable Speedup with Threads on an IBM SP2 (A - 1 processor, B - 2p, C - 4p, D - 6p)



(b) Speedup With Threads on 3 PC's & 10 Mb/s Ethernet, (A - 1p, B - 3p)

**x-axis: objects in torus** $= N^2$

**y-axis: runtime in seconds**

Figure 7: Speedup Using Threads

## 7 CONCLUSIONS

Most PDES systems support the active-server world view even though many models are more easily expressed with the active-process world view. ParaSol supports the active-process world view by implementing simulation processes with user level threads. ParaSol's single stack thread system avoids the memory fragmentation usually present in the unused portion of a thread's stack. ParaSol uses the copy of a thread's state made when the thread is removed from the shared stack as a checkpoint buffer. Finally, ParaSol hides the details of interprocessor communication from the user through a novel thread migration mechanism. Efficient use of threads is one reason that PARASOL can speedup the simulation of parallel models that are as simple to code as sequential models.

## REFERENCES

Carson, Jim. 1993. Modeling and simulation worldviews. *Proceedings of the 1993 Winter Simulation Conference*, 18–23.

Jefferson, D. R. 1985. Virtual Time. *ACM Transactions on Programming Languages and System*, 7:3:404–425.

Fujimoto, R. 1989. Time warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6:3:211–239.

Martin, D. E., and T. J. McBrayer. 1997. *Warped – a parallel discrete event simulator (documentation for version 0.8)*. Dept. of EECS, University of Cincinnati, OH.

Bagrodia, R. L. 1991. Iterative Design of Efficient Simulators Using Maisie. *Proceedings of the 1991 Winter Simulation Conference*, 243–247.

Gomes, F., S. Franks, B. Unger, and Z. Xiao. 1995. Simkit: a high performance logical process simulation class library in C++. *Proceedings of the 1995 Winter Simulation Conference*, 706–713.

Booth, C. J. M., and D. I. Bruce. 1997. Stack-free process-oriented simulation, *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, 182–185.

Mascarenhas, E., F. Knop, and V. Rego. 1995. ParaSol: a multi-threaded system for parallel simulation based on mobile threads. *Proceedings of the 1995 Winter Simulation Conference*, 690–697.

Schwetman, H. D. 1986. CSIM: a C-based process-oriented simulation language. *Proceedings of the 1986 Winter Simulation Conference*, 387–396.

Kleiman, D., D. Shah, and B. Smalders. 1996. *Programming with Threads*. Englewood Cliffs, NJ: Prentice-Hall.

Mascarenhas, E., and V. Rego. 1996. Ariadne: architecture of a portable threads system supporting thread migration. *Software - Practice and Experience*, 26:3:327–356.

## AUTHOR BIOGRAPHIES

**REUBEN PASQUINI** is a graduate student in computer science at Purdue University. His research interests include parallel and distributed simulation.

**VERNON REGO** is a Professor of Computer Sciences at Purdue University. He was awarded the 1992 IEEE/Gordon Bell Prize in parallel processing research, and is an Editor of *IEEE Transactions on Computers*. His research interests include parallel simulation, parallel processing, modeling and software engineering.