Proceedings of the 1995 Winter Simulation Conference
ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman

ParaSol: A MULTITHREADED SYSTEM FOR PARALLEL SIMULATION BASED ON MOBILE THREADS

Edward Mascarenhas
Felipe Knop
Vernon Rego

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

ABSTRACT

ParaSol is a novel multithreaded system for shared- and distributed-memory parallel simulation, designed to support a variety of domain-specific Simulation Object Libraries. We report on the design of the ParaSol kernel, which drives executions based on optimistic and adaptive synchronization protocols. The active-transaction flow methodology we advocate is enabled by an underlying, efficient lightweight process system. Though this process- and object- interaction view is known to both simplify and speed transition from model design to simulation implementation, migratable threads and objects pose many serious challenges to efficient kernel operation. Good solutions to these challenging problems are key to good simulator performance. We present techniques for the support of optimistic parallel simulations, addressing synchronization, state-saving, rollback, inter-process communication, and process scheduling.

1 INTRODUCTION

Simulations that progress in time via a discrete sequence of events – discrete event simulations – are challenging targets of parallelization. The challenge lies in moving a parallel simulation forward to completion as fast as possible in real time, while satisfying simulation-time related synchronization constraints. We have embarked on the construction of a four-tiered software architecture, called the *ACES* system (Knop et al. 1995), for parallel computation and simulation applications. In this paper, we focus our attention on the *ParaSol* layer, which supports general parallel simulation and computation through the process- and object- interaction view.

We resort to standard parallel simulation terminology (Fujimoto 1990) to present background and design ideas. If a physical system to be simulated can be viewed as a system of interacting *physical* processes, a simulator for such a system consists of interacting *logical* processes (LPs), each progressing from one event to the next in simulation time, where such time is tracked by a local clock and called the *local virtual time* (LVT). To simplify discussion, we will assume a one-to-one correspondence between physical and logical processes. Dynamic entities in the physical system may move from one physical process to another, and are represented by active-transactions (threads) that flow between LPs in the logical system. By binding simulation time-stamps (denoting when transaction-related events occur) to transactions, this information flow enables communication – and thus synchronization – between LPs.

By allowing cooperating processors to synchronize, a parallel simulator eliminates invalid simulation trajectories generated by *causality errors*. A causality error is said to occur at an LP if this LP finds itself in violation of the fundamental simulation rule: *events must be processed in order of non-decreasing time*. The major focus of parallel simulation research centers around implementation and performance assessment of two well-known synchronization protocols: the *conservative* protocol and the *optimistic* protocol. In the *conservative* approach (Chandy and Misra 1979), events are executed strictly in order of occurrence in simulation time. In the *optimistic* approach (Jefferson 1985), an LP processes events as event messages become available, hoping the physical order of event message arrival corresponds to order of nondecreasing time. A causality error occurs if an event arrives at an LP with a time-stamp value that is less than the LVT of the LP – thus motivating use of the term *straggler* to describe the event message – rendering the processing of the LP potentially invalid. When this occurs, the computation is rolled back and restarted from a previously saved and error-free state. *State-saving* and *rollback* mechanisms enable implicit LP synchronization.

Software interfaces to sequential simulation applications tend to support one or more of three well-known world views: *event-scheduling*, *activity-scanning* and *process-interaction*. *ParaSol* is designed to support *active-transaction* flow in process-interactive simulations. This approach to modeling is used by many popular sequential simulation languages (e.g., CSIM, GPSS, SIMAN). The idea here is to develop a process-style description of a transaction's activity as it flows through a system: In the alternate, *active-resource* view, transactions are passive entities that are processed by active resources, i.e., process-style descriptions implement resource functions: In Figure 1 is shown two code segments highlighting the difference between these approaches.

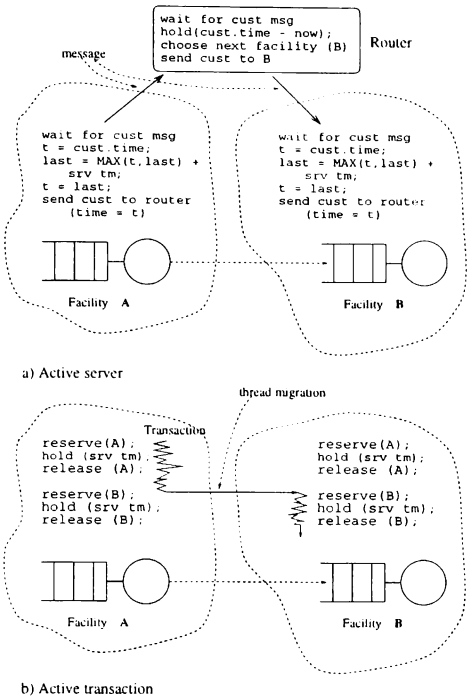

Figure 1: Program code using (a) the active-resource view, and (b) the active-transaction view. In the first case customers are represented by messages, and resources by processes. In the second, customers are represented by processes, and resources by objects.

1.1 Simulation Software and *ParaSol*

The *ParaSol* system was motivated by concerns of ease-of-use – for enabling experimentation, generality – for domain-specific parallel computations, exploitation of threads – for efficiency, and rapid transitioning, and portability – for maximum usability. Code development in C++ offers the benefits of flexibility and extensibility. Internal components are modified and/or replaced with little difficulty to support application-specifics – a result of inheritance features of C++. For example, *ParaSol* binds to an arbitrary parallel programming library like PVM (Sunderam 1992), Conch (Topol 1992) for remote process spawning and message-passing.

To the best of our knowledge, *ParaSol* is the first *run-time threads* based parallel simulation system. Here, transactions are implemented via *time-stamped threads* which transparently *migrate* between processors to access model resources (resident on distinct processor memories). Based on experimental work reported by Sang et al. (1993), the use of mobile threads has several advantages: locality of reference, potential for load balancing, one-time transmission, simplicity in application-level coding, etc. On the other hand, the complexity of working with threads instead of simple messages is significant: issues complicated by threads include LP scheduling, state saving and rollback, interfacing simulator kernel and threads systems etc. Finally, a key advantage of our approach is ease of programming – an easily overlooked but beneficial feature, motivated by the apparently low use of parallel simulation software in the simulation community. This advantage is simply a result of three design decisions: support for active-transaction flow, optimistic synchronization as an extreme form of adaptive synchronization, and cheap, mobile threads.

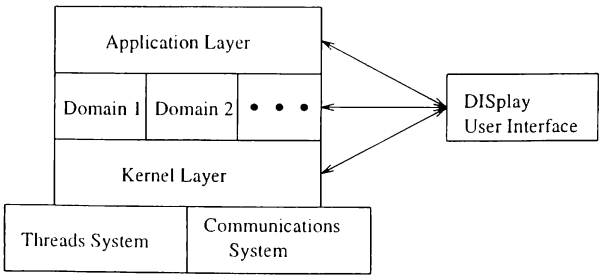

Figure 2: Layering in *ParaSol*

A few parallel simulation systems have been developed in the past decade, some as experimental, research-oriented systems, and others as commercial-grade systems (Jefferson and Bellenot 1987, Baezner et al. 1990 ,Bagrodia 1991, Steinman 1992, BoyanTech Inc. 1994).

2 ARCHITECTURE

The *ParaSol* system consists of three layers, shown as the top three layers in Figure 2. Support layers beneath these three include a threads layer and a communications layer. An additional, but independent, *DISplay* library supports visualization functions at all layers. The *ParaSol kernel* is essentially the parallel simulation engine, responsible for driving all the other modules (these are described briefly in Section 3). Kernel modules access threads-layer primitives and communications layer primitives, protecting application-level codes from system details.

The *ParaSol* kernel is currently layered upon the *Ariadne* (Mascarenhas and Rego 1995a) threads system. Like other threads systems, *Ariadne* supports dynamic creation and destruction of threads, priority-based scheduling and context switching. But *Ariadne* is novel in that it also supports thread migration between processors, user-customized schedulers, thread-check-pointing, and thread-image restore operations. The *Ariadne Parallel Programming System* (Mascarenhas and Rego 1995b) is based on the notion of migratable threads. With support from *Ariadne*, *ParaSol* adopts transaction-migration as a primary remote-object access mechanism, exploiting transparent migration of threads through supporting object location/relocation mechanisms.

Since *ParaSol* was motivated by domain-specific methodologies, the domain layer consists of distinct domain libraries. For example, the *queueing domain* contains functionality (operations on servers and queues) that is different from functionality provided by a *particle-physics* domain (grid definition,

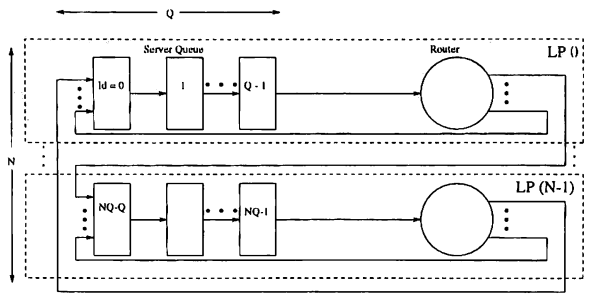

Figure 3: Closed Queueing Network with N Switches and Q Servers per Switch

cluster generation, particle tracing). Thus, each domain library provides an interface to resources that are specific to a particular domain. Domains that provide more functionality correspondingly relieve the user of programming detail at the application level. A user may select domain-specific functions that suit the application, thus eliminating nontrivial code redesign. Typical domains include switching systems, particle physics, manufacturing systems, digital logic circuits, and combat simulations.

2.1 The Application Layer

User-level code is developed using functions provided in the domain and kernel layers (see Table 2). An executing *ParaSol* application will consist of a main program (a Unix process) which simultaneously executes on multiple user-specified processors/machines. Each such (Unix) process is called a *ParaSol process* and hosts one or more logical process (LP) threads, one or more *global* objects associated with the LP thread (the number and type of such objects depends on the application), and threads that represent active-transactions. *Global* objects are objects whose presence and location are known to all LPs. Function `main` binds LPs to *ParaSol* processes at run-time, first creating the requisite number of LP threads, and then scheduling them for execution. When an LP thread receives control, it creates global objects (these are specified by the modeler) and registers them with a *controller* process (a *ParaSol* process with added responsibilities), so that these objects may be accessed by other LPs.

2.2 Example: A Closed Queueing Network

We provide a simple *ParaSol* model of a closed queueing network (CQN) using the queueing domain library. Code developed in Maisie (Bagrodia 1991) for the same example, was based on the active-resource approach. In contrast, the *ParaSol* code developed here is based on the active-transaction approach. Consider a system of N fully connected switches, each hosting Q single-server (fifo) queues in tandem. Customers arriving at a switch are served by each of these Q servers in sequence, and then routed, uniformly randomly, for another round of service to one of the N switches. Initially, each switch is assigned J jobs.

```
1   const int N = 5;            /* number of switches */
2   const int NSERV = 10;       /* servers per switch */
3   const int NJOBS = 10;           /* number of jobs */
4   const int NTRIPS = 10;         /* number of trips */
5   PSOL s;                     /* simulation manager */
6   FACILITY lServer[NSERV]; /* models local servers */
7   FACILITY rServer[N];    /* models remote servers */

8   main() {
9       s = new PSol();
10      s->simulate();
11  }

12  LPtoHOSTMapper(void) {  /* binds LPs to processes */
13      int i, numProcs;
14      numProcs = s->getNumProcs();
15      for (i = 0; i < N; i++)     // N queues
16          s->bindLP(TQUEUE, i%numProcs, MEDIUM, 3, 0,
                    0, i, NSERV, NJOBS);
17  } /* end LPtoHOSTMapper */

18  LP TQUEUE(int qid, int Q, int J){ /* tandem queue */
19      int j, n, q;

20      for (q=0; q < Q; q++) /* create local servers */
21          lServer[q] = new Facility(qid*Q+q, FIFO);

22      for (n=0; n < N; n++)/* create remote servers */
23          if (qid != n)
24              rServer[n] = newRemoteObject(n*Q, Facility);
25          else
26              rServer[n] = lServer[0];

27      for (j=0; j < J; j++)          /* create jobs */
28            s->create(jobThread, MEDIUM, 4, 0, 0, j, i
                      qid, Q, NTRIPS);

29      s->waitTermination();    /* wait for a signal */

30      for (q=0; q < Q; q++)    /* print statistics  */
31          lServers[q]->print_report();
32  } /* end TQUEUE */

33  ATHREAD jobThread(int jid, int qid, int Q, int T) {
34      int trips = 0;                /* models a job */
35      int q;
36      float rno;
37      int n = qid;

38      while (trips < T) {
39          for (q = 0; q < Q; q++) {
40                  if (q == 0)/*reserve and use server */
41                      rServer[n]->reserve();
42                  else
43                      lServer[q]->reserve();
44                  lServer[q]->hold(expon(MSRVTIME));
45                  lserver[q]->release();  /* release  */
46          }
47          n = urand(0, N-1)*Q; /* destination switch */
48      }
49  } /* end jobThread */
```

Figure 4: Implementing a Closed Queueing Network in *ParaSol*

Table 1: *ParaSol* Kernel Primitives

Transaction Primitives	
int trCreate(void (*funcname)(...), int stacksize, args);	Create a transaction to execute `funcname` with a stack of size `stacksize` and arguments `args`
int trMyid();	Return the identifier of the transaction
int trHold(double time);	Suspend execution of the transaction for `time`
int trSuspend(void);	Suspend execution of transaction indefinitely
int trResume(int tid, double delta_t);	Resume a suspended transaction at lvt+ `delta_t`
int trCancel(int tid);	Cancel a scheduled execution of a transaction `tid`
int trMigrate(int LPid);	Migrate a transaction to LP `LPid`
void trExit(void);	Finish this transaction
int trSetAttrib(int index, void* buf, int size);	Set transaction attribute at `index` to `buf`
int trGetAttrib(int index, void* buf, int size);	Get transaction attribute from `index` into `buf`
Logical Processes	
void bindLP(void (*func)(...), int procId, int stksz, args);	Bind an LP to a process `procId` and create an LP thread to execute `func` in process `procId`
int lpMyid();	Get the identifier of the LP
Objects	
void *newRemoteObject(int id, ObjectClass);	Create a dummy object of type `ObjectClass` and identifier `id`. Locate the object
int objRegister(GlobalObject *objPtr);	Register object for state saving
int objDeregister(GlobalObject *objPtr);	Deregister the object from state saving
Miscellaneous	
void simulate(void);	Initialize and start the simulation
int getNumProcs(void);	Returns the number of Unix processes in the system
void printf(char *fmt, ...);	Output a message at the current time
void waitTermination(void);	Wait until termination
void signalTermination(void);	Signal a termination

Simulation ends when each job completes some user defined number of trips around the network. Service times are assumed to be i.i.d exponential random variables at all servers. Figure 3 depicts a CQN with N switches, and Q tandem queues at each switch. A complete *ParaSol* program for this application is shown in Figure 4.

The main function initializes the *ParaSol* system and begins simulation via a call to the `simulate()` method (lines 9 and 10). This ParaSol method makes a call to a user written `LPtoHostMapper()` function whose task is to create all LPs and bind them to (Unix) processes (line 16). The `bindLP()` kernel primitive creates an LP thread at the process specified in its `procId` argument. In the example, each LP `TQUEUE` creates Q servers (modeled by the type `Facility` from the queueing domain) (lines 20 and 21). Each LP also needs to know the location of the first server at each of the N switches, since transactions must migrate to LPs hosting these servers. This is obtained by a call to function `newRemoteObject()` on line 24. Note that each LP and each global object possesses a unique system-wide identifier through which it is accessed. In the example, LPs are numbered from 0 to $N-1$, and server objects are numbered from 0 to $NQ-1$. After each LP thread creates `NJOBS` transactions (line 28), thus assigning the requisite number of jobs to each switch, it relinquishes control by invoking function `waitTermination()`.

Transactions representing jobs are active, and may migrate between LPs. Each `jobThread()` makes `NTRIPS` trips to the switches. Each server is reserved for service by a job via a queueing-domain function `reserve()`. If the server is busy, the job is queued. When service begins, upon a job's return from function `reserve()`, the server is made to remain occupied for some amount of (simulated) time via the function `hold()`. The server is finally released via function `release()`, so that other jobs may use the server (lines 39 – 46). After obtaining service from Q consecutive servers, a customer must decide whether to continue receiving service at another switch, or simply terminate execution. If it decides to continue, it generates a uniformly distributed random integer in $[0, N-1]$ to locate a destination switch (line 47). Observe that the corresponding Maisie model effects this through a special `router` entity. In *ParaSol* this routing is implicit in the call to `reserve` (line 41), since this function knows where (i.e., on which processor) the server to be reserved is located.

3 KERNEL MODULES

In this section we address the functionality of key modules in the *ParaSol* kernel, providing insights into algorithms and data structures used to implement various tasks. This prepares the ground for issues discussed in the following section. Each *ParaSol* module is developed as one or more classes in C++. Figure 5 summarizes critical components of the kernel in pictorial form.

The kernel is interfaced to five major layers: the domain and application layers (above), the threads and communications layers (below), and the DISplay

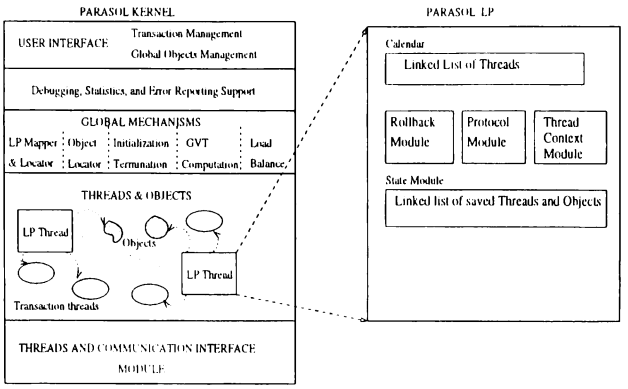

Figure 5: The *ParaSol* Kernel

layer. Functionality in the *ParaSol* kernel is deliberately kept to a minimum – to implement core simulation primitives. As a result, kernel design, implementation, and maintenance remain relatively simple. The effect is to drive all complex application-related functionality into higher system layers, and in particular into the domain layer.

The kernel modules that interface with the threads and communications systems consist of a base class, that defines the interface, and several derived classes. Depending on which derived class is used, appropriate threads systems and communications systems are bound to the kernel – this provides portability between software environments.

The communications interface module in the kernel has two important functions. First, it allows *ParaSol* to be used with any arbitrary parallel programming system selected by the user. Second, it allows the *ParaSol* system to execute in sequential or in parallel mode. In the sequential mode, message passing and other actions related to parallel simulation are disabled. A crucial optimization is made possible for simulation on clusters of shared-memory multiprocessors: the communication module exploits native inter process communication (IPC) mechanisms (e.g., Unix message queues) for message passing.

The remaining kernel modules may be considered to be either Global mechanisms or Logical Process mechanisms. LP mechanisms are local to an LP and control the simulation of transactions associated with the LP.

3.1 Global Mechanisms

The major global mechanisms are shown in Figure 5. Here we discuss mechanisms that impact the behavior of transactions in *ParaSol*. The LP Mapper and Locator maps LPs to processes and maintains the location of all LPs in the system. The global object locator mechanism is similar to the LP locator and provides the location of any object, given its unique identifier. The location of an object is the identifier of the LP at which the global object is created. The *global virtual time* (GVT) module in *ParaSol* computes the GVT periodically. The algorithm is similar to the GVT algorithm used in TWOS (Jefferson 1985). To account for transactions in transit we piggyback acknowledgements onto messages. Two distinct modules are responsible for system initialization and termination. Initialization occurs when the simulation "driver" and associated modules are created. At this time the threads system is also initialized, and multiple processes are initiated by the communications system.

3.2 Local LP Mechanisms

A *ParaSol* kernel may manage an unlimited number of LPs (see Figure 5) simultaneously. In doing so, the kernel provides a critical form of efficiency: distinct LP threads resident within a single process may freely pass data to one another (shared memory). Further, by placing several LPs within a single process, rollback may be significantly reduced (since LPs within a process are scheduled strictly by minimum time-stamp of transaction). For each LP resident in a given host, an internal LP object is created. This object contains LP related data (LVT, transaction with minimum time-stamp, etc.) and pointers to various modules.

The most important structure contained in an LP is the Calendar: this structure stores *transactions* that have executed in the past, and transactions scheduled for future execution. The calendar also records side effects caused by execution of transactions, allowing these effects to be undone, when and if necessary. In *ParaSol* a calendar entry consists of a time-stamp and a pointer to a suspended thread context. Thus a transaction is merely a thread context coupled with a time-stamp. The Thread Context Module stores time-stamped contexts of active and inactive threads in the system. Maintaining inactive contexts is necessary because of the potential for rollbacks – inactive contexts may need to become active and recompute. The primary data structure used is a hash queue indexed by a thread identifier and thread activation time. The hash queue allows rapid access to the context of any transaction in the LP given only the thread identifier.

The State Module in *ParaSol* is responsible for saving LP-state at some user-specified frequency, depending on run-time characteristics. System state in *ParaSol* is defined by the state of all threads (LP and active-transaction threads) and objects (associated with either LPs or active transactions). Thus, saving system-state in *ParaSol* requires saving both *data (objects)* and *computations (threads)*. This is in contrast to existing parallel simulation systems, where only data is saved. Objects that are local to threads (local variables and data structures) are automatically saved – as local thread state – when a thread is saved. But objects that are global need explicit save actions. State saving in *ParaSol* is transparent at the (user) application-level. However, global objects defined not at the domain layer but at the application layer must be registered for state-saving through functions provided in the kernel.

State-saving overheads are known to have serious

impact on the performance of optimistic protocols. Minimizing such overheads, whenever possible, is critical. *ParaSol*'s state-saving algorithm for threads operates incrementally and infrequently. The latter is a result of intermittent check-pointing, where frequency is determined by a user-specified event count. State-saving is incremental because instead of saving all threads within an LP at a checkpoint, only threads that have run within the last interval are saved.

The Rollback Module contains algorithms for effecting rollback: an LP's state is rolled back from its LVT t_n to its state at some past time t_p. The coast-forwarding phase (Fujimoto 1990) is necessary when state is saved infrequently, as is done in *ParaSol*. During the coast-forward, user commands are executed but kernel primitives execute code selectively. For example, transactions do not (re)migrate during the coast-forward.

4 THREADS AND TRANSACTIONS

Use of threads-based active-transactions in optimistic parallel simulations provides definite advantages, while presenting nontrivial design problems that must be resolved. In the rest of this discussion, we focus on the implications of thread usage for transaction implementation in *ParaSol*.

4.1 Basic Thread Primitives

Threads systems support lightweight processes: low-overhead processes ideal for process-based simulations. Creation costs and context-switching costs of such processes are significantly lower than corresponding costs for heavyweight processes. All threads systems support creation, context-switching, suspension, resumption, and destruction of processes These primitives are sufficient for the implementation of process-interactive simulators in sequential and parallel environments. One way of using threads in process-based simulations is to make the threads system an integral part of the simulator. This method is used in CSIM (Schwetman 1986), a sequential process-based simulation system written in C. A serious disadvantage of this approach is that the threads system cannot be replaced if required – for instance, when an improved threads system becomes available. Certain *ParaSol* requirements may not be easily met by other threads systems: support for a large number of co-existing threads, and thread migration. Because *Ariadne* supports threads in user-space, threads may co-exist in large numbers. One of *Ariadne*'s main design goals was the efficient support of process-based parallel simulators (Mascarenhas and Rego 1995a).

4.2 Scheduler

Every simulator must utilize a scheduler to schedule events: the earliest event in the system is scheduled for immediate execution. In process-based simulations, the scheduler selects the transaction (thread) with the lowest time-stamp and gives it control – since a thread handles processing of its own events. Most

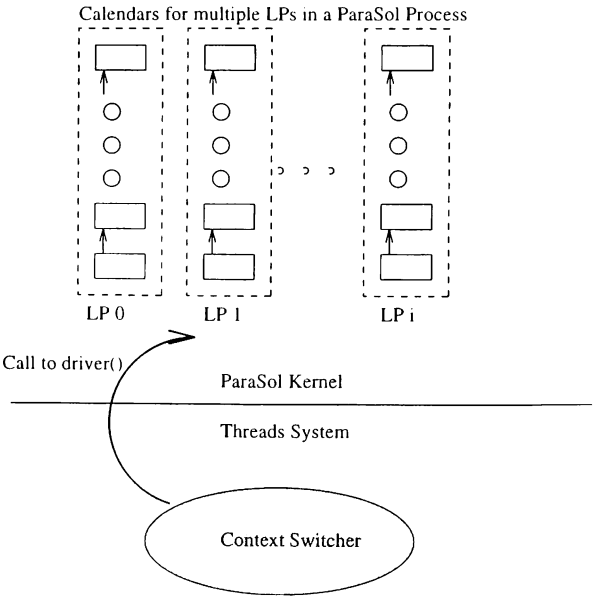

Figure 6: *ParaSol* as a Thread Scheduler for Ariadne

threads systems (including *Ariadne*) offer built-in, OS style priority-based schedulers that are not appropriate for simulations. *Ariadne* permits scheduler customization at the user level; the user creates scheduler functions that are installed at run-time. When a custom scheduler is installed, the calls to *Ariadne*'s built-in scheduler are mapped onto calls to user-installed functions. These functions must implement the required scheduling policy. As shown in Figure 6, a threads system may view a *simulator as merely a customized scheduler*. *Ariadne*'s context-switch mechanism invokes the *ParaSol* driver to obtain the next thread to run.

Ariadne's methodology allows thread contexts to be stored in user space. An appropriate data structure can be used to store thread contexts. *ParaSol* makes use of a linked list based calendar, and a hash queue table, to store and retrieve contexts. The *ParaSol* driver retrieves the thread context of the next transaction to execute from the linked list calendar, and returns it to the threads system.

4.3 Transaction Migration

In parallel simulations based on the active-resource approach, messages are used to send passive entities (such as jobs, customers) between resources. Messages may also be used for communication in transaction-flow based systems (e.g., Maisie). In contrast, our approach is based on *active* transaction flow: we believe that migrate processes is a very natural representation of active transaction flow. *ParaSol* implements this transaction flow via thread migration – a direct mapping that brings a model as close to the physical system as possible. Transaction flow is necessary for transactions to access resources in various parts of the system.

```
extern PSol* _pSol;  /* the simulation controller */
class Facility: public GlobalObject {
    void *data;   /* pointer to the actual object */
 public:
    Facility();
    void reserve(void);
    ...
};

void Facility::reserve(void)
{
     int lpId;
     int facId;

     if ((lpId = getLP_Id()) !=
          _pSol->getCurrentLP_Ptr()->getLPid()) {
          facId = getObjId();   /* save id of objt */
          _pSol->trMigrate(lpId);/* migrate remote */
          /* this code runs on remote processor */
          (PSmapObjMgr->existingObject(facId)))->reserve();
     }
     else {
         /* normal processing - the facility is local */
         ...
     }
}
```

Figure 7: Example of Migration: `reserve()` Primitive of Queueing Domain

Migration entails moving a transaction from one processor to another. *Ariadne* provides the low-level thread migration support required by *ParaSol*. A thread, consisting of a *thread context area* (tca) and stack, is "packed" into a buffer and sent to the destination via communications layer primitives. At the destination, the message is "unpacked" into a thread-shell and is ready for execution. The cost of migrating a thread in this manner is equal to the cost of sending and receiving a message of the same length (Mascarenhas and Rego 1995a). This point is of great significance, because *ParaSol* does not curtail the number of migrations. The transaction-flow approach may result in a large number of threads, each of which can migrate. Since migrant threads are responsible for rollback, efficient migration is important. Details on migration in *Ariadne* can be found in (Mascarenhas and Rego 1995a). Transmitting threads instead of messages adds marginal overhead in the form of stack information. But since the dominant component of transmission time is due to transmission latency and not message size or even data packing/unpacking, thread-migration costs are equivalent to message transmission costs (Sang et al. 1993). Finally, migration does not always involve message transmission because both the source and destination LPs may reside within the same process or in the same shared memory multiprocessor.

As an example of the use of migration, consider the implementation of the `reserve()` primitive from the queuing domain. This primitive allows a transaction to reserve a server in a `Facility` (see the closed queueing network example in Figure 4). If the `reserve()` primitive is executed when the server is busy, execution of the transaction is suspended until the server becomes available. When the `reserve()` primitive returns, the server is ready to serve the reserving transaction. In Figure 7 is shown a C++ code-segment implementing this primitive. First, a check is made to determine whether the `Facility` to be reserved is hosted by the current LP. If so, transaction execution proceeds locally. If not, the object identifier is saved in local variable `facId`, and the kernel's migrate primitive is invoked. As indicated earlier, the `Facility` object contains the identifier of its host LP. The `trMigrate()` primitive calls *Ariadne*'s migrate primitive. When `trMigrate()` returns, the transaction is at the destination LP; here it accesses the `Facility` locally. At the destination LP, the pointer to the object is updated by a call to `existingObject()` (using the saved value of *facId*), provided within the Object Locator module. This is followed by a recursive call to `reserve()` using the updated pointer; the call is guaranteed to proceed, since the object is now local to the LP.

A thread migration capability is not essential for implementing process-based simulations. But with this capability, parallel simulation users are protected from having to use explicit send and receive data messages in application code. As shown in the closed queueing network example, migration makes the source code for the parallel simulation almost identical to sequential simulation, since all accesses to remote data become local after migration. Moreover, migration is a direct form of active-transaction flow, resulting in a powerful threads-based modeling abstraction.

4.4 Transaction State

Optimistic parallel simulations save system state to support rollback. State saved in *ParaSol* includes object-state as well as thread-state. The threads system must support save and restore facilities for thread contexts: the `context_save()` kernel primitive causes a thread's tca and essential stack to be saved in a buffer (a thread image), with a buffer address returned. To save the state of all the threads in an LP, this primitive must be called for each active thread run since the last check point. A thread is restored by placing its saved image (tca, stack) into its current context, achieved via a call to the `context_restore()` kernel primitive. The cost of state-saving and restoration is directly proportional to the size of a thread's stack. This size is a function of the number and size of local variables declared within a thread and its degree of nesting at a point of suspension.

4.5 LP Migration

LP migration, a special case of thread migration, is an important facility for load balancing. When an LP migrates, all objects and threads associated with the LP migrate along with it. This also includes the calendar, thread context module, state module, and other data structures used by the LP. Because of the availability of a basic thread migration facility, the simultaneous migration of a batch of threads does not

pose a problem. Migrating all objects in an LP is made possible with the aid of functions to save the state of each object at the sender and to recreate the object at the receiver. Migrating objects in a heterogeneous environment entails the use of network-independent formats. With additional help from the LP locator and Object locator modules, the locations of LPs and Objects – which may change as a simulation proceeds – *ParaSol* makes dynamic load balancing feasible.

5 CONCLUSION

The *ParaSol* system is an experimental test-bed for studies in domain-specific parallel stochastic simulation. At the present time, the major kernel modules have been tested and initial experimentation is in progress. Domain layers are presently under construction. Based on simple experiments we have already conducted, we conclude that the proposed approach is feasible. More importantly, the system is easy to use and experiment with. Modelers that are adept at sequential transaction-flow based simulation should find the *ParaSol* approach simple. Further, porting existing sequential applications to *ParaSol* will be possible with low effort. Domain layers under construction at this time include a queueing domain, a particle-physics domain, and an adaptive quadrature domain. The kernel currently supports optimistic synchronization; this is currently being extended to support adaptive synchronization. Support for load-balancing, fault-tolerance, and replication is also in progress. We plan to report on system performance when domain layers are ready.

ACKNOWLEDGEMENTS

This research was supported in part by ONR-9310233, NATO-CRG900108, and ARO-93G0045. The second author was supported by CNPq-Brazil process number 260059/91.9.

REFERENCES

Baezner D., G. Lomow, and B. Unger. 1990. Sim++: The Transition to Distributed Simulation. *Distributed Simulation, SCS Simulation Series*, 211–218.

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

BoyanTech, Inc. 1995. *CPSim 1.0 User's Guide and Reference Manual.* BoyanTech, Inc., McLean, VA 22102.

Chandy K. M. and J. Misra. 1979. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Trans. on Softw. Eng.*, 5(5):440–452.

Fujimoto R. 1990. Parallel Discrete Event Simulation. *CACM*, 33(10):30–53.

Jefferson D. and S. Bellenot. 1987. Distributed Simulation and the Time Warp Operating System. *ACM Operating System Review*, 77–93.

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

Knop F., E. Mascarenhas, V. Rego, and V. Sunderam. 1995. Fail-Safe Concurrent Simulation with *EcliPSe*: An Introduction. *Simulation Practice & Theory (to appear).*

Mascarenhas E. and V. Rego. 1995a. *Ariadne*: Architecture of a Portable Threads System Supporting Thread Migration. *Software-Practice and Experience (to appear).*

Mascarenhas E. and V. Rego. 1995b. Migrant Threads on Processor Farms: Parallel Programming with Ariadne. Technical report in preparation, Computer Sciences Department, Purdue University.

Sang J., E. Mascarenhas, and V. Rego. 1993. Process Mobility in Distributed-Memory Simulation Systems. In *Proceedings of the 1993 Winter Simulation Conference*, 722–730.

Schwetman H. 1986. A C-based Process Oriented Simulation Language. In *Proceedings of the 1986 Winter Simulation Conference*, 387–396.

Steinman J. S. 1992. SPEEDES: A Unified Approach to Parallel Simulation. In *Proceedings of 6th Workshop on Parallel and Distributed Simulation, Simulation Series*, 75–84.

Sunderam V. S. 1990. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339.

Topol B. 1992. Conch: Second Generation Heterogeneous Computing. Technical report, Department of Mathematics and Computer Science, Emory University.

AUTHOR BIOGRAPHIES

EDWARD MASCARENHAS is a Ph.D. student in Computer Sciences at Purdue University. He received a Masters degree in Industrial Engineering from NITIE (Bombay, India), and a Masters degree in Computer Sciences from Purdue University (West Lafayette) in 1993. His research interests include parallel computation, distributed simulation, and multithreaded programming environments.

FELIPE KNOP is a Ph.D. student in Computer Sciences at Purdue University. He received a Masters degree in Computer Sciences from Purdue University in 1993 and a Masters degree in Electrical Engineering from University of São Paulo, Brazil, in 1990. His current research interests include parallel and distributed simulation, and multiprocessor operating systems.

VERNON REGO is a Professor of Computer Sciences at Purdue University. He received his M.Sc.(Hons) in Mathematics from B.I.T.S (Pilani, India), and an M.S. and Ph.D. in Computer Science from Michigan State University (East Lansing) in 1985. 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 and software engineering.