Proceedings of the 1996 Winter Simulation Conference
ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain

A PARALLEL GPSS BASED ON THE PARASOL SIMULATION SYSTEM

Felipe Knop
Edward Mascarenhas
Vernon Rego

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

ABSTRACT

Much of the research in parallel discrete-event simulation (PDES) has resulted in new experimental simulation languages or toolkits. Meanwhile, the simulation community continues to use existing (serial) commercial tools which are reportedly more powerful and flexible from a modeler's point of view. A possible way to increase the impact of PDES in the simulation community is to make existing simulation packages execute in parallel. Towards this end, we present a parallelization of the GPSS simulation language. We implement parallel GPSS as a GPSS-to-C++ translator and execute the transformed code with the help of the *ParaSol* parallel simulation system. The mapping from GPSS to *ParaSol* is simple because, unlike other parallel simulation systems, *ParaSol* is transaction oriented. On the other hand, because GPSS was not designed with parallelism in mind, there are GPSS constructs that can behave poorly in a parallel environment. We present details on the mapping, some of the challenges we faced in this task, and key solutions that we adopted to enhance parallelism.

1 INTRODUCTION

It has been said that research in parallel discrete-event simulation (PDES) has not made an impact on the simulation community (Fujimoto 1993). A possible reason for this is that extant PDES systems typically sacrifice model implementation effort for the promise of speedup. Application programmers are forced to use explicit message exchange constructs, lookahead information, and even state-saving/restoration procedures. Meanwhile, the simulation community continues to use existing (serial) commercial tools, which are reportedly more powerful and flexible from the modeler's point of view.

In this paper we describe a project to parallelize the GPSS simulation language (Bobillier et al. 1976). GPSS was chosen because of its widespread use and potentially large base of existing simulation programs. Since GPSS was not developed with parallel execution in mind, it offers a good test of the suitability of current PDES work for the parallelization of sequential simulation languages.

This work is based on the *ParaSol* project, aimed at developing a parallel simulation system based on mobile threads (Mascarenhas et al. 1995). *ParaSol*'s main design goals are ease of use – to minimize user-visible complexities of parallel simulation – and flexibility – to maximize the system's use across different application domains. We implement parallel GPSS as a GPSS-to-C++ translator and execute the transformed program using *ParaSol*. The mapping from GPSS to *ParaSol* is simplified because both are transaction oriented. But though, at first glance, both systems appear to have many similarities, there are dramatic differences as well. Because GPSS was not designed to run on a parallel machine, it has features that can make it behave poorly in a parallel environment. We present details on the mapping, some challenges that we faced in parallelizing GPSS, and some key solutions that we adopted in the implementation.

The remainder of the paper is organized as follows. Section 2 contains a review of related work on parallelizing sequential simulation languages. Sections 3 and 4 present overviews of *ParaSol* and GPSS, respectively. In section 5 we describe the most important aspects of the mapping from GPSS to *ParaSol*. In Section 6 we point out some problems caused by the sequential nature of GPSS, and in Section 7 we describe our solutions to these problems. Performance experiments are presented in Section 8. We conclude in Section 9.

2 RELATED WORK

Today's popular (sequential) simulation languages were not designed for parallel execution and, as a result, are difficult to parallelize. It should come as no surprise, therefore, that research in PDES systems invariably results in new simulation languages or toolkits. For examples of such systems see Bagrodia (1991), BoyanTech (1995), and Gomes et al. (1995). Typically, these systems offer an application interface consisting of a set of logical processes that exchange timestamped messages with one another, using either language constructs or function calls.

Tsai and Fujimoto (1993) describe a framework for parallelizing simulation languages. In this framework, a common layer implements a set of basic primitives that can be used to build compilers for existing languages. The common layer maintains two abstractions, *state variables* and *unprocessed events*, and standard operations that can be applied to these. For example, variables can be created, destroyed, read, and modified, and events can be scheduled, examined, and deleted. The parallel implementation of the shared variables and their operations is achieved through the use of *space time memory*, allowing variables to be accessed by entities that execute at different points in simulation time. The parallel implementation of the event list is obtained with mechanisms similar to those found in Time Warp. As a case study, the paper presents an implementation of the SIMSCRIPT II.5 language for a shared-memory environment. It is unclear whether the approach can be used to parallelize transaction-oriented languages such as GPSS for architectures that do not support shared-memory multiprocessing.

Nicol and Heidelberger (1994) describe a different approach. Here, the goal is to parallelize simulation tools with little or no modification, because these tools are generally large and complex. The idea is to use an existing tool to define sub-models that will run on different processors, with tool *extensions* defining interfaces between sub-models. The extensions incorporate all required communication and synchronization. Despite the apparent simplification, this implies that a sequential application will have to undergo some modification in order for it to run in parallel. In implementing tool extensions, no information about the sub-models is assumed, except that which is provided explicitly by the sub-model. This constrains parallel models in that they are forced to run under a conservative synchronization mechanism. The computation and dissemination of lookahead information – sometimes beneficial in conservative mechanisms – is encapsulated in the extensions. The proposed approach is applied to produce a library which extends the CSIM simulation system (Schwetman 1986). To enable extensions, without requiring significant changes, a tool will generally require features that may not be present in other simulation tools.

3 *ParaSol* OVERVIEW

ParaSol is a parallel simulation system based on the (active-transaction) process-interaction paradigm. Instead of using timestamped messages for communication and synchronization among logical processes (LPs), as done in existing systems, *ParaSol* relies on transactions that transparently migrate between LPs, to obtain a more powerful effect. Indeed, this transparency leads to greatly simplified model development for many simulation problems, and was an important design consideration.

ParaSol presents a programming environment that offers *transactions* – dynamic, computational units with some private data – and a set of *global objects*. Both transactions and objects are distributed among the physical processors hosting a simulation. Transactions, which are dynamically created and destroyed, usually spend their time either performing local computations or accessing objects. To access an object located at a remote process, a transaction *migrates* to the process where the object is located, thus enhancing locality. A transaction causes simulation time to pass when it executes a *hold* command (analogous to the ADVANCE block in GPSS).

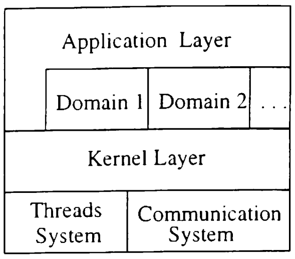

Figure 1: *ParaSol*'s Layered Architecture

ParaSol's architecture is shown in Figure 1. The kernel provides basic services, and the domain libraries support higher level services geared towards specific application domains. The kernel insulates the upper layers from most parallel simulation details, including transaction management, migration, communication, rollback, etc. The kernel is supported from below by the *Ariadne* threads system (Mascarenhas and Rego 1996) and a suitable communications subsystem, e.g., PVM (Sunderam 1990). Support for migratable threads is provided by the Ariadne system.

The kernel programming interface is represented by public methods of class `PSol`, the main simulation class in *ParaSol*'s C++ interface. An explanation of some of the kernel primitives will help clarify the description of the mapping from GPSS to *ParaSol*.

A transaction's execution environment consists of the local variables of the function currently being executed, and the local variables of all functions in its calling chain. Transactions are directly supported by the kernel layer through the use of threads. They are dynamically created with method `trCreate()`. As in other process-oriented simulation languages, simulation time progresses when transactions execute `trHold(double time)`, a primitive analogous to `hold()` in CSIM. If the simulation time is t before `trHold(x)` is executed, it will be $t + x$ after this primitive is executed. Other transactions may execute between time t and time $t + x$.

Transactions are suspended and resumed with primitives

```
int trSuspend(void)
int trResume(int tid, double delta_t)
```

If a transaction is to be suspended, layers above the kernel are responsible for storing the transaction's id, so that it can be used when the transaction is resumed.

In *ParaSol* the term logical process (LP) – slightly twisted in relation to the definition used in other systems – describes the *static* (albeit still active) part of the simulation model. While transactions migrate from LP to LP, LPs themselves remain largely static. Each logical process in *ParaSol* consists of a single thread – the "LP thread" – and objects hosted by the LP. An LP is created and bound to one specific (UNIX) process at the start of the simulation, using primitive `bindLP()`.

Transactions (threads) migrate from one LP to another, using primitive `trMigrate(int LPid)`. Migrations occur instantaneously in terms of simulated time; if a delay is required it can be added by invoking `trHold()`.

To make a C++ object usable as a *ParaSol* global object, two kernel primitives are invoked on its behalf. The first registers the object for state saving, causing methods `save()` and `restore()` to be called when the system is saving or restoring state (in this instance the kernel does not insulate the layers above from parallel simulation functions). The second primitive registers the object as a global object, allowing transactions throughout the system to locate the position (LP number) of the object. Global objects have one "original" copy, located at some process, and several proxies. In general there is one proxy at each of the other processes. When a transaction accesses a proxy's method, a check is made within the method to determine if the object is an original or a proxy. If the object is a proxy, the transaction migrates to the appropriate LP where the same method is invoked on the original object.

4 GPSS OVERVIEW

Like *ParaSol*, GPSS adopts an active-transaction process-interaction world view. A GPSS program consists of a sequence of "blocks" that represents the flow of transactions through the system. Transactions, which are created by the GENERATE block, progress from one block to the next unless routed to another block by a TRANSFER block.

A large number of permanent entities are provided to facilitate modeling. For example, *Facilities* are FIFO single-server queues. Block SEIZE is used by a transaction to "seize" a facility's server. Seizing an already seized facility forces a transaction to *block* until the facility is released. A transaction holding a facility releases it using RELEASE. *Storages* are similar to facilities but may hold more than one transaction at a time.

Transactions have access to local (per transaction) variables called "parameters" and global (system-wide) variables called "savevalues". There is also support for computational entities such as random number generators. Time advances through the execution of the ADVANCE block, which suspends the invoking transaction for a user-specified amount of simulated time (only integer numbers allowed). Transactions finish their execution by entering the TERMINATE block. At each such block, a global termination counter is decremented by some given amount. When the counter reaches 0, the simulation is terminated.

Some powerful features make GPSS useful in manufacturing applications. For example, the SPLIT, MATCH, GATHER, and ASSEMBLE blocks allow transactions to split into two or more transactions and later rejoin, effectively enabling the modeling of machine parts that split and take different paths.

```
GENERATE   FN$ARRIV
QUEUE      WAIT
SEIZE      BARB
DEPART     WAIT
ADVANCE    FN$SERV
RELEASE    BARB
TERMINATE  1
START      10
```

Figure 2: Example of GPSS Program

A small example of a GPSS program is given in Figure 2. This example (taken from Bobillier et al. (1976)) models a barber shop: a barber, represented by facility BARB, can serve only one customer at a time. Customers, represented by transactions, enter the shop and wait for service if the barber is busy. Customers are served in order of arrival. A GPSS queue—an entity that helps in generating queueing statistics—called WAIT is used to represent the waiting room. The START block sets the termination counter, indicating how many transactions must terminate for the simulation to finish. The transactions are created by the GENERATE block, with the time between creations given by function "ARRIV".

At the heart of the GPSS simulator we find *transaction chains* containing all transactions in the system. The *future events chain* (FEC), actually a simulation calendar, contains the list of transactions scheduled for execution at a future time, in general those that have entered an ADVANCE block. The *current events chain* (CEC) contains transactions due to execute at the current time, but also contains those that are blocked, waiting for conditions such as the release of a facility. The CEC is organized as a priority queue, with transactions having higher priority coming first. Among transactions with the same priority, a FIFO ordering is adopted. The CEC is always scanned completely in each simulation cycle, from beginning to end, before it receives new transactions from the FEC. The transactions in the CEC chain are executed until they enter ADVANCE or become blocked for some reason. Certain events, such as the release of a facility or storage, force the simulation engine to rescan the whole CEC, since some transactions on the CEC may have become ready to execute once again.

5 GPSS-*ParaSol* MAPPING

At a first glance, GPSS appears readily amenable to execution support from *ParaSol*. We implement the language as a GPSS-to-C++ translator, where the transformed program invokes primitives from the *ParaSol* kernel and a "GPSS domain" library.

Entities in GPSS are mapped as follows. Static entities such as facilities, queues, storages, and even savevalues are implemented as *ParaSol* global objects. As an example, the GPSS facility object, similar but not identical to the Facility object in *ParaSol*'s queueing domain, presents the outside world with methods such as `seize()` and `release()`, which implement the behavior of the SEIZE and RELEASE blocks, respectively. Other methods are provided to return facility statistics (e.g., facility utilization and queueing delay). These methods are required not only for printing final statistics but also for implementing GPSS's *standard attributes*. The GPSS facility object also contains state saving and restoration methods which are required by *ParaSol*'s optimistic synchronization protocol.

Not surprisingly, GPSS's transactions are actually *ParaSol*'s transaction threads. The execution path of GPSS transactions, which comprise the bulk of the GPSS input lines, is incorporated into the thread code. Threads then move from block to block as directed by the input program. Transaction attributes such as parameters and priorities are implemented via the *thread attribute area* supported by *Ariadne*. An alternative would be to place these transaction attributes on the thread's stack, but this would hinder the translation of GPSS blocks that allow one transaction to examine another's attributes. Time advances (ADVANCE block) are achieved through calls to the kernel `trHold()` primitive.

Transactions entering a SEIZE block, for example, are represented by threads executing

```
GPSS_Facility_ptr->seize()
```

If the transaction is "refused" by the SEIZE block because the facility is already in use, then the corresponding thread is placed in a queue inside the GPSS facility object. In terms of the domain-kernel interface, the queue is nothing more than a list of thread ids. The thread is suspended with the kernel's `trSuspend()` primitive. To resume the thread, it suffices to call the kernel's `trResume()` primitive with the thread id as parameter. When resumed (by another thread executing the RELEASE block), the thread eventually returns from `seize()`, now owning the facility.

Parallel execution of GPSS is achieved by first distributing the global objects among different LPs and then placing these LPs in distinct processes, as illustrated in Figure 3. This figure shows a transaction that executes a method at a proxy facility at LP #1. This forces it to migrate to LP #2 and access the same method at the real facility. The number of LPs used in the program is determined at compile time. This number gives an upper bound on the maximum parallelism possible. The assignment of LPs to processors is done at run time, based on the number of processors available; each processor may host more than one LP. When a thread accesses a remote object, the thread *migrates* to the LP where the object is located, as explained in Section 3. For example, inside `GPSS_Facility::seize()` (and also in many of the GPSS domain library methods), there is a test to determine whether the facility is remote. If it is, the thread migrates to the appropriate LP. Otherwise, the thread goes through the actual SEIZE procedure on the local object. A thread migrates along with its attribute area, which is required for the correct implementation of parameters, priorities and other GPSS transaction attributes.

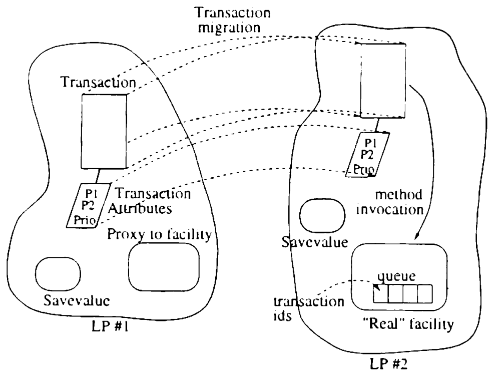

Figure 3: Transactions and GPSS Objects

```
ATHREAD generateBlk5(void)
{
  int intTmp;
  gpssP->advance(8);
  for(; ;) {
    pSolP->trCreate(transactionThread,
                          <stk sz> ...);
    intTmp = uniform_AB(40,60);
    if(intTmp > 0)
      gpssP->advance(intTmp);
  }
}
```

Figure 4: *ParaSol* Code Created for Block GENERATE 50,10,8

A GPSS program may have several GENERATE blocks, with each possibly specifying a different rate of creation for transactions. For each GENERATE block in the program, the translator creates a *generate thread*. Figure 4 shows an example of code executed by a generate thread. The example corresponds to the sequence GENERATE 50,10,8, which is supposed to create new transactions with inter-creation time uniformly distributed between 50-10 and 50+10 time units, with the first one created at time 8. In the figure, method `advance()` is nothing but a wrapper around `trHold()`. The reason for not having `trHold()` itself in the code is explained in Section 7.4. The generate threads, one for each GENERATE block, are created at the beginning of the simulation.

Termination requires the use of a centralized `GPSS_Termination` global object. Threads executing the TERMINATE block migrate to the LP holding the `GPSS_Termination` object, using the same mechanism as in `GPSS_Facility`. There, the termination counter, which is a member of the termination object, is decremented; when it gets to 0 all processes are informed about the termination. When TERMINATE's operand is 0, the termination counter need not be decremented. As an optimization for this situation, the thread does *not* migrate to the termination object's location, since there is no need for it to access the termination object.

6 ISSUES INVOLVED IN PARALLELIZING GPSS

The main obstacle in parallelizing GPSS is the many instances where the language definition, and its natural sequential implementation, assumes the existence of centralized data structures. Typical examples are statements such as

```
        TEST E    Q8,15               (1)
        TRANSFER BOTH,HERE,THERE      (2)
VAR     VARIABLE Q1+Q2+Q3             (3)
```

In the first statement above, the transaction must block while the number of elements in queue number 8 is not 15. In a typical GPSS implementation, the transaction is left in the CEC (see Section 4) and is awakened every time the CEC is rescanned. This kind of blocking, called a "non-unique blocking condition", is one where the system does not know a priori what can be done to unblock the transaction. Consequently, there is no specialized queue for "transactions waiting for queue 8 to have 15 elements".

In the second statement, according to the language definition, the transaction must attempt to go to the block labeled HERE and, if the block is "refusing transactions" (for example, the block is a SEIZE and the facility is in use), the transaction must try the block labeled THERE. If the latter also refuses entry, the transaction must suspend itself inside the TRANSFER block and try again when the CEC is rescanned.

In the third statement, the transaction must obtain the number of elements in queues 1, 2, and 3 to compute the value of the variable.

To be able to parallelize GPSS, we must do away with centralized data structures such as the FEC and CEC. While the FEC is handled "automatically" by the PDES synchronization mechanism, the CEC still requires some suitable replacement. The existence of the CEC may severely impact performance, since it assumes that all transactions it holds will have to be rescanned every time some "significant" event, such as the release of a facility, occurs. While the rescan in a sequential setting is already recognized as inefficient (Bobillier et al. 1976), doing it in a distributed environment would certainly lead to bottlenecks: if the same scheme is adopted in a distributed implementation, the "significant" events would have to be reported to all processes, resulting in unacceptable communication cost.

Statement 3 above highlights the importance of object placement in the performance of parallel GPSS: if the three queue objects are located on different processors, then the thread implementing the transaction must migrate at least twice just to compute the value required in the example.

Statement 2, which like statement 1 presents a non-unique blocking condition, illustrates how much the language depends on centralized information: if the objects referred to by blocks HERE and THERE are located at different processes, then the transaction would have to migrate twice to determine the status of these blocks. Moreover, it would have to repeat the check every time some "significant" event occurs at any place in the system.

Another difficulty related to the CEC is its policy of considering transaction priorities when inserting transactions in the chain. Like many other systems, *ParaSol* does not have a CEC-equivalent. It inserts transactions in the calendar based only on their resumption times.

7 SOME SOLUTIONS

7.1 Object Placement

A judicious placement of objects may alleviate some performance problems by reducing the number of thread migrations. For instance, in statement 2, having the objects accessed by blocks HERE and THERE in the same process would help tremendously, as would having queues 1,2, and 3 in the same process at statement 3. Of course, applying this reasoning to all the blocks would result in a sequential simulation! It can be observed that having queues 1–3 in different processes is likely to be less damaging than having objects in HERE and THERE in different processes, since in the latter case the set of migrations may occur several times for each transaction that enters a TRANSFER BOTH, ... block.

Since, in some applications, the user may be better equipped than the compiler to decide the location of some key entities, constructs are provided to allow manual placement of entities at either compile time or program load time.

7.2 Replacement of the CEC

The CEC and the mechanism used to handle non-unique blocking conditions must be altered for the parallel implementation. Suppose a transaction is blocked, waiting for the condition "queue 4 has 3 elements and storage 6 is full". In the sequential implementation, the transaction is placed on the CEC. Every time the CEC is rescanned, the transaction is resumed. The transaction then proceeds to check one more time if the two conditions are true. In the distributed setting, queue 4 and storage 6 may be located on different LPs, say LP1 and LP2, respectively.

Since a "global" CEC is not maintained, the following scheme is adopted when a transaction is about to block on a non-unique blocking condition (we use the example above to explain it):

- Thread T1 visits queue 4 at LP1 and places a notification record [T1,LP3] at queue 4
- Thread T1 then visits storage 6 at LP2 and places a record [T1,LP3] at storage 6
- Thread T1 moves to LP3 and suspends itself. LP3 may be LP1, LP2, or yet another LP.
- When the status of queue 4, for example, changes, the queue object (that is, the particular thread which, executing some method in the queue object, alters the queue's status) checks its notification records. Upon finding the entry [T1,LP3] there, it sends a messenger thread to LP3. Once there, the messenger confirms that T1 is still blocked and, if so, awakens it. T1 then visits queue 4 and storage 6 once again.

Although still expensive, this scheme is nevertheless cheaper than sending broadcast notification messages every time any entity changes state (the distributed equivalent of the CEC). For unique blocking conditions, the implementation is simpler and more efficient: transactions are suspended and placed in existing queues at the appropriate entities. As an example, consider the queue of transactions waiting to enter the facility entity, located within the GPSS facility object.

7.3 Space-Time Memory

Besides triggering expensive multiple migrations, constructs such as that shown in example 3 above (`Q1+Q2+Q3`) create havoc with optimistic synchronization protocols, since migrations may cause rollbacks at one or more LPs. On the other hand, in such constructs the queue objects themselves are not altered by the operations following the migration. Therefore, the following scheme may be adopted:

- Variables that are frequently accessed in a read-only fashion, such as the queue size in the given example, are stored in a "space-time" manner, not unlike the Tsai and Fujimoto approach (1993): not only the most current value, but also *all* past values are stored. An appropriate format for this is a time-ordered sequence of (t_i, v_i) pairs, where v_i is the variable's value and t_i is the simulated time at which the variable is loaded with the value v_i.
- If a thread migrates to an LP only to retrieve the value of some variable, then a special "read-only" migration is used that does not cause a rollback even if the destination LP is already in the future of the migrated thread. In this case, the migrated thread is scheduled as if it were not a straggler, and it then accesses a past value of the desired variable by examining the past history of values stored in the space-time memory.

To avoid causality problems, the migrating thread should refrain from altering the state of any object. If it decides to do so, the thread is required to update its migration status from "read-only migration" to the default "read-write" migration. Then, if the destination LP is in the future of the arriving thread, in terms of virtual time, a rollback procedure is immediately initiated.

The mechanism used for reclaiming the space-time memory is similar to that used in fossil collection: values saved before GVT (Global Virtual Time, Fujimoto 1990) may be discarded since they are no longer needed. Notice, however, that all manipulation of the space-time memory is done at the domain level, independently of the kernel. The kernel could have provided all support for space-time memory by incorporating it into the state saving mechanisms. Past values of variables would then be retrieved from system snapshots taken for state saving purposes. The drawback of this approach is that it would require the kernel to save state at *every* execution of an event, preventing the use of infrequent state saving mechanisms.

If keeping all values since the GVT in space-time memory is too expensive, the space-time memory may be configured to store only the last few values. This may be the case if each value requires a large amount of memory, or if the GVT is not computed frequently enough, If some older value that is not stored in space-time memory is requested, then the read-only migration is updated to the default migration.

Together with the read-only migration, the space-time memory can be used in the implementation of several GPSS constructs, including the savevalues and many of GPSS's "standard attributes".

7.4 Transaction Priorities and the CEC

It is possible to simulate the structure of the CEC (transactions ordered by insertion time and priority) without having to change *ParaSol*'s kernel. First, we observe that the *ParaSol* calendar, the equivalent of GPSS's FEC and CEC combined, already respects a FIFO ordering for threads having the same time: if a thread executes `trHold(x)`, and the current time is t, its resumption is scheduled as the *last* entry in the calendar having time $t + x$. Next, we can take advantage of GPSS's *integer* time advances and use *ParaSol*'s arbitrary time advances. If the thread priority changes from 0 to p $(0 \leq p < 128)$ then the next `ADVANCE x` issued by the thread at time t will cause the thread to be resumed at *ParaSol* time $t + x - cp$, where $c < 1/128$ is a constant. In this way, the thread is resumed before all threads with priority smaller than p scheduled for $t + x$ (GPSS time).

The simulation of the CEC's structure is not perfect, though. The order of scheduling two transactions arriving at an LP from two different processors, with the same simulation timestamp, is essentially random, and depends on the real times at which the transactions arrive at the LP.

8 EXPERIMENTS: ENTITY MAPPING

We report results of initial performance experiments that measure the effect of the entity-to-LP mapping.

Our test program simulates a closed queueing network and is written in a GPSS subset. The network has 64 facilities, divided into four groups of 16 facilities in tandem. Jobs that leave one group can be routed to any other group. The GPSS code is shown in Figure 5. The `GENERATE` block at line 2 generates 32 transactions, with inter-generation time uniformly distributed between 80 and 120. These transactions begin execution at the line following the `GENERATE` block. Once created, transactions go with equal probability to blocks labeled `SWT1`, `SWT2`, `SWT3`, or `SWT4`. At each such block, transactions visit 16 facilities in series, after which they go back to the `TRANSFER` block at line 3. Blocks at lines 31–33 actually define the termination condition for this particular program: a transaction is created at time 160000. Upon being created at line 31, it executes the termination block

```
 1        SIMULATE   1
 2        GENERATE   100,20,,32  32 jobs generated
 *                   once every U(80,120) units
 3 ROUT TRANSFER     0.250,,SWT1 With prob 1/4,
 *                   transfer transactions to blocks
 *                   SWT1, SWT2, SWT3, or SWT4
 4        TRANSFER   0.333,,SWT2
 5        TRANSFER   0.500,,SWT3
 6        TRANSFER   ,SWT4
 7 SWT1   ASSIGN     1,16    P1 <- 16
 8        ASSIGN     2,1     Entities [1-16]
 9 LOO1   QUEUE      P2
10        SEIZE      P2
11        DEPART     P2
12        ADVANCE    50,10   Service time: U(40,60)
13        RELEASE    P2
14        ASSIGN     2+,1    P2++
15        LOOP       1,LOO1  P1--;if(P1==0) goto LOO1
16        TRANSFER   ,ROUT

17 SWT2   ASSIGN     1,16    P1 <- 16
18        ASSIGN     2,17    Entities [17-32]
19 LOO2   QUEUE      P2
20        SEIZE      P2
21        DEPART     P2
22        ADVANCE    50,10   Service time: U(40,60)
23        RELEASE    P2
24        ASSIGN     2+,1    P2++
25        LOOP       1,LOO2  P1--;if(P1==0) goto LOO2
26        TRANSFER   ,ROUT
27 SWT3   ASSIGN     1,16    P1 <- 16
28        ASSIGN     2,33    Entities [33-48]
          [...]
29 SWT4   ASSIGN     1,16    P1 <- 16
30        ASSIGN     2,49    Entities [49-64]
          [...]
31        GENERATE   160000  create transctns every
 *                           160000 time units    (*)
32        TERMINATE  1       termination transctn (+)
33        START      1       set termination cnt to 1
34        END
```

Figure 5: A Closed Queueing Network Coded in the GPSS Subset

at line 32. Since the termination counter is now 0, the simulation is terminated.

How GPSS entities are mapped onto LPs is crucial to the performance of parallel GPSS. We have experimented with different mappings, in order to evaluate its effect on execution time. The ideal mapping for the given queueing network is to have each group of 16 tandem facilities and queues placed at a different LP: facilities 1–16 are placed at LP 0, facilities 17–32 at LP 1, and so on. Queues are treated identically. By doing this, we balance computation across processes. Also, we minimize communication, since transactions visit 16 facilities and queues at a time without the need for remote object access.

Like queues and facilities, GENERATE blocks and termination objects must also be mapped. All transactions created by GENERATE begin their execution at the LP hosting the generate thread (Section 5). Transactions executing TERMINATE with a non-zero operand must migrate to the LP that hosts the termination object. The GENERATE block on line 31 should be placed at the LP that contains the termination object. LP 0 is chosen for both. The generate block on line 2 can be placed at any LP, and LP 0 is chosen for this block.

Table 1 shows the results of executions using different entity-to-LP mappings. All four processors in a SUN SPARCstation 20, a shared memory multiprocessor, were

Table 1: Performance of a Closed Queueing Network Coded in GPSS with Several Different Entity-to-LP Mappings

	Mapping	Exec time	Num rollbacks	Msg read ops
1	Optimal mapping	52.7	1959	14907
2	GENERATE block (*) and TERMINATE block (+) in different LPs	53.5	2976	18515
3	Facilities and queues 13–16 at LP 1; facilities and queues 29–32 at LP 0	66.4	4755	33779
4	Facility and queue 10 at LP 1	66.6	3953	26819
5	Unbalanced: facilities and queues 17–32 at LP 0 (LP 1 hosts no entities)	82.3	4424	35653

used in the experiment. Execution times are in seconds. Approach 1 shows, as a reference, the performance of the optimal mapping. Approach 2 shows the performance of a near-optimal mapping; only the GENERATE block at line 31 is moved to LP 1. This seems to be a trivial variation of the optimal mapping, since only one transaction is created on line 31. In an optimistic environment, however, the sequence of events is more complex. Since all job transactions are generated at LP 0, LP 1 is initially idle and has no alternative but to schedule the only calendar entry it has, which is for the generate thread (line 31) at time 160000. Once scheduled, the generate thread creates a transaction that immediately migrates to LP 0 in search of the termination object. At LP 0, this transaction is placed in the calendar but not scheduled, since time at LP 0 is still far from the termination time. The GENERATE block on line 31 creates a transaction *every* 160000 time units, so that a new transaction is created at each of the virtual times 320000, 480000, etc.; each immediately migrates to LP 0. Eventually, job transactions migrate from LP 0 to LP 1, causing a rollback at LP 1. Several anti-thread messages (*ParaSol*'s equivalent of anti-messages) are sent from LP 1 to LP 0. The whole process is repeated whenever LP 1 finds itself idle. Though the effect on execution time, as shown in Table 1, is small, the effect on number of rollbacks is not. On a distributed-memory environment, execution times would suffer because of the larger communication delays.

Approach 3 shows a more significant departure from the optimal mapping: four facilities and queues are moved from LP 0 to LP 1, and vice versa. Not surprisingly, the execution time, number of rollbacks, and number of message operations are all higher than in the optimal mapping. In approach 4, the optimal mapping is changed by placing only one facility-queue pair (number 10)—originally at LP 0—at LP 1. The execution time is close to that given by approach 3. When compared to the optimal mapping, this indicates that even small changes may have a noticeable effect on execution time. After accessing facilities 1–9, transactions at LP 0 must migrate to LP 1 to access facility 10, and then return to LP 0 to access facilities 11–16. These extra migrations account for the increased communication/rollback overhead over the optimal mapping.

Finally, approach 5 shows a more radical departure: all entities at LP 1 are now assigned to LP 0, resulting in an unbalanced load. The execution time is greatly affected, with the processor hosting LP 1 becoming idle for the entire run.

9 CONCLUSION

This paper describes the implementation of a parallel GPSS simulation language, based on the *ParaSol* system. Because of GPSS's widespread use, this effort is one significant step towards making parallel simulation more practical and useful to the simulation community. *ParaSol*'s process-orientation enhances the mapping from GPSS to *ParaSol* , since GPSS is also process-oriented. GPSS, however, relies heavily on centralized data structures. These can hamper the performance of parallel applications. We describe some techniques to tackle bottlenecks arising in the parallel GPSS implementation, such as user-guided placement of simulation objects, the use of notification mechanisms to control communication costs in non-unique blocking conditions, and the use of space-time memory and read-only migrations to reduce the number of rollbacks.

At the present time, the implementation of a GPSS core is complete. Although we expect GPSS programs to run in parallel without changes, some programs may not possess a sufficient amount of inherent parallelism. These may require code rearrangement or may run efficiently only for select entity-processor mappings.

ACKNOWLEDGMENTS

This research was supported in part by ONR-9310233, BMDO-34798-MA, and NSF-CCR 9311862. The first author was supported by CNPq-Brazil, grant 260059/91.9.

REFERENCES

Bagrodia, R.L. 1991. Iterative design of efficient simulations using Maisie. In *Proceedings of the 1991 Winter Simulation Conference*, pages 243–247.

Bobillier, P., B. Kahan, and A. Probst. 1976. *Simulation with GPSS and GPSS V.* Prentice Hall.

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

Fujimoto, R.M. 1990. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53.

Fujimoto, R.M. 1993. Parallel discrete event simulation: Will the field survive? *ORSA Journal of Computing*, 5(3):213–230.

Gomes, F., S. Franks, B. Unger, Z. Xiao, J. Cleary, and A. Covington. 1995. SimKit: A high performance logical process simulation class in C++. In *Proceedings of the 1995 Winter Simulation Conference*, pages 706–713.

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

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

Nicol, D. and P. Heidelberger. 1994. On extending parallelism to serial simulators. Technical Report ICASE Report No. 94–95, Institute for Computer Applications in Science and Engineering – NASA Langley Research Center. December.

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

Sunderam, V.S. 1990. PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4).

Tsai, J.-J. and R.M. Fujimoto. 1993. Automatic parallelization of discrete event simulation programs. In *Proceedings of the 1993 Winter Simulation Conference*, pages 697–705.

AUTHOR BIOGRAPHIES

FELIPE KNOP, Ph.D., Computer Sciences Department, Purdue University, (August 1996), 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.

EDWARD MASCARENHAS, Ph.D., Computer Sciences Department, Purdue University, (August 1996), 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.

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, modeling and software engineering.