Proceedings of the 1988 Winter Simulation Conference
M. Abrams, P. Haigh, and J. Comfort (eds.)

The Object Library for Parallel Simulation (OLPS)

Marc Abrams
Distributed Systems Group
Department of Computer Science
Stanford University
Stanford, CA 94305

Abstract

The Object Library for Parallel Simulation (OLPS) is a set of C++ objects from which a parallel, discrete event simulation or simulation language can be built and run on multiprocessor workstations. OLPS provides a common interface to the simulation programmer for several parallel simulation algorithms, including implementations of the Chandy-Misra and Time-Warp algorithms. OLPS also provides instrumentation to compare the performance of each parallel algorithm on the same simulation problem. The system currently runs on the V distributed system on the DEC Firefly.

Keywords: parallel simulation, object-oriented programming, Chandy-Misra simulation algorithm, Time-Warp simulation algorithm

1 MOTIVATION

This paper describes the initial implementation of and performance measurements from a system for parallel simulation called the *Object Library for Parallel Simulation* (OLPS). The system is targeted to multiprocessor workstations, and is suitable for general purpose, discrete event simulation.

A number of techniques have been proposed in the literature for carrying out simulation in parallel, for example [3, 7, 24, 4, 5, 8, 10, 11, 14, 18, 23]. OLPS, rather than implementing a new simulation technique, provides a framework within which a variety of proposed techniques are and can be implemented.

The objectives of OLPS are to:

1. provide the simulation programmer with implementations of several parallel simulation algorithms, as well as a common interface to these algorithms,
2. provide instrumentation to measure the run-time behavior of the selected simulation algorithm,
3. factor out common operations among the techniques (e.g., interprocess communication, message routing, flow control, random number generation) and provide a single implementation for these, and
4. facilitate experimentation through:
   (a) modifying existing algorithms, and
   (b) adding new algorithms to the library.

OLPS is not, however, intended to be a simulation language. Rather, we expect that OLPS will provide the run-time support needed to port existing simulation languages to multiprocessors.

OLPS currently implements two algorithms: Chandy-Misra [5] with deadlock avoidance and Time-Warp [11]. The Object Library is implemented in C++ using the Argonne National Library macro package [16] to permit its porting to various operating systems. Our implementation runs on the V operating system [6] on a five processor DEC Firefly [22].

One motivation for OLPS is that performance measurements reported so far indicate that parallel simulation can be highly efficient for some problems but slower than single-processor simulation in other cases. For example, Reed, Maloney, and McCredie report that the Chandy-Misra algorithm achieves near linear speedup for tandem queues, while for central server queueing networks the algorithm usually runs faster on a single processor than on multiple processors. Speedup as used here is the ratio of completion time of a parallel simulation program on an N processor trial to a 1 processor trial. Until more is understood about parallel simulation performance, a practical parallel simulation system will need to offer multiple techniques with a common interface. The simulation programmer can then experiment with which algorithms work best in particular problem domains.

Another motivation for OLPS is to compare through measurement alternate algorithms. Comparing different algorithms on the same benchmark always leaves open the question of whether the differences observed are due

to the way in which each algorithm is implemented or are due to the inherent differences in the algorithms themselves. OLPS facilitates such comparisons, because one benchmark simulation program can be run with alternate parallel algorithms via the common interface, and because the alternate algorithms share the same implementation of common services. This eliminates one source of implementation differences.

The structure of OLPS is presented next. §3 describes measurements of its performance. §4 discusses related work.

2 OBJECT LIBRARY FOR PARALLEL SIMULATION

Objects (or *classes* in C++ [21]) consist of data and a set of operations that modify the data. For example, OLPS contains an object implementing a communication channel (the CHANNEL object), in which the data is contiguous memory shared by processes and the operations allow concurrent processes to read and write the data.

A CHANNEL and other objects that rely on no other objects are called *base* objects. All other objects are *derived* objects that *inherit* the data and operations of other base or derived objects. For example, OLPS contains a SINGLELINKEDLIST base object. A discrete random variable with arbitrary distribution is implemented by the ARBITRARYRANDOMVARIABLE object, which is derived from a SINGLELINKEDLIST by storing the distribution function in the list and adding the operation `Sample` which calls a random number generator before accessing the distribution list.

2.1 The OLPS Object Hierarchy

One may represent the hierarchy of base and derived objects in the form of a tree. Figure 1 illustrates the overall organization of OLPS: A set of base objects implements mechanisms common to all simulation techniques. From these objects implementing the Chandy-Mis ra algorithm are derived by adding mechanisms to generate and respond to null messages. From the base objects a set of objects implementing the Time-Warp algorithm is derived.

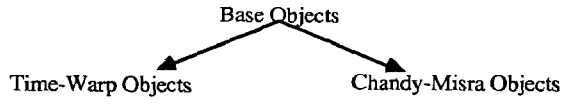

Figure 1: Overal Object Library Structure

To understand the actual objects used in OLPS, it is first necessary to understand the simulation model used by the Chandy-Misra and Time-Warp. The system to be simulated is viewed as a finite set of *physical processes* (e.g., the terminals, disks, memory, and CPU of a computer system). Each physical process is associated with a set of events (e.g., an I/O request arrives at a disk). Physical processes interact by sending and receiving messages. A simulation program then consists of a set of logical processes (the operating system processes) corresponding to the physical processes, which can "predict the exact sequence of message transmissions in the physical system [17]."

The OLPS Object Library defines an object called NODE, which corresponds to a physical process. (As described later, a NODE object is assigned during simulation to a logical process, which is then scheduled to run on a processor.) A node exports the following operations:

```
CreateNode(Id, Type, Sequencer*,
   Responder*, Router*)
DestroyNode()
void SetRoute(Id, Node*)
void PreLoad(Node*, int NoJobs, ...)
Msg* Input()
Msg* RespondTo(Msg*)
void Output(Msg*)
```

All objects in the library are implemented as containers, storing pointers to objects, rather than the objects themselves. Thus formats for important data structures, such as the text of messages, are defined by the user. The list of exported operations contains asterisks to denote pointers to objects, in the style of C++.

A complete simulation program instantiates one node for each physical process (via `CreateNode`) and sets its output route (via `SetRoute`). Nodes may also be created or destroyed dynamically during execution by calls to `CreateNode` and `DestroyNode`. `CreateNode` also sets two node attributes: a type and a name. Both are unsigned integers chosen by the user. `PreLoad` sets the initial condition of a NODE.

A NODE object can receive messages (via `Input`), simulate them to produce an output message stream (via `RespondTo`), and route the output messages to their destination nodes (via `Output`).

Figure 2 shows the basic organization of the Object Library. The NODE object is derived from the following four objects:

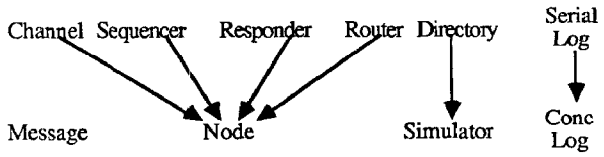

Figure 2: More Detailed Object Hierarchy

SEQUENCER: Implements the Input operation of a NODE object. Collates the multiple input streams to a node in time-stamp order. Decides in what order arriving messages are presented to the RESPONDER. The library currently contains two types of responders: FIFO (first in, first out), and LIFO (last in, first out).

RESPONDER: Implements the RespondTo operation of a NODE object. The user always derives an object from the RESPONDER to performs the actual simulation of the corresponding physical process.

ROUTER: Implements the Output operation of a NODE object. Different router objects are provided to route based on different rules. Currently the library contains a probabilistic router (PROBROUTER), a router used when there is only one output route (DECFREEROUTER), and a router that deletes messages (SINK).

CHANNEL: Implements the Input and Output operations of SEQUENCER and ROUTER objects, respectively. Provides a communication medium and its protocol. There are currently two implementations available. The CQCHANNEL provides a monitor to access object CIRCULARQ, which is a circular queue of fixed, user specified size. The LINKLISTCHANNEL is a monitor accessing a linked list.

Figure 3 illustrates the relationship of the four objects from which a NODE is derived. Messages arriving from multiple sources first enter a CHANNEL object, which contains a spin lock to serialize access to it. Messages then pass to the SEQUENCER, RESPONDER, and ROUTER objects before leaving the NODE.

Different versions of these objects exist in the library to implement different parallel simulation algorithms (e.g., Chandy-Misra, Time-Warp), different routing rules, different sequencing rules, and so on. Thus by proper selection of these during NODE object instantiation via CreateNode, a user may quickly construct a variety of simulations.

The remaining objects in Figure 2 are:

DIRECTORY: DIRECTORY is only used during simulation initiation and termination. Contains a pointer to all NODE objects in the system. Exports operations to map node names to pointers and to find all nodes of a given type. One key operation exported (Assign) assigns nodes to operating system processes.

SIMULATOR: Uses DIRECTORY to assign nodes to operating system processes, and to assign processes to processors. Exports operation Simulate, which is called by the user to initiate execution of all NODE objects.

Operation Simulate creates as many processes as there are nodes. Each process then executes:

```
Node* Me = Assign();
// assign a node to this process
for (int N=0; N<EventLimit; N++)
    Me->Output(
        Me->RespondTo(Me->Input()));
```

(The termination of node execution after it has executed EventLimit number of events is a simplification of the actual termination mechanism.)

SERIALLOG and CONCLOG: These log events by storing them in a circular queue of user specified size in memory during simulation execution. SERIALLOG is used by a single process. CONCLOG adds a monitor to control access to a SERIALLOG by multiple processes. After execution, the log is written to disk.

MESSAGE: Contains a type (e.g., User, Suspend, Annihilate), a source and destination node name, a time stamp, and a pointer to user text of user specified format. Derived from type MESSAGE is type LLMESSAGE, which additionally contains link fields for use with the LINKLISTCHANNEL.

2.2 Complete Simulation Program

Figure 4 illustrates the process by which a user assembles a complete simulation program using the Object Library.

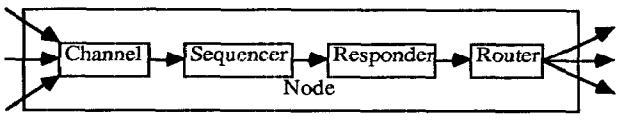

Figure 3: Relationship of Objects Comprising a Node

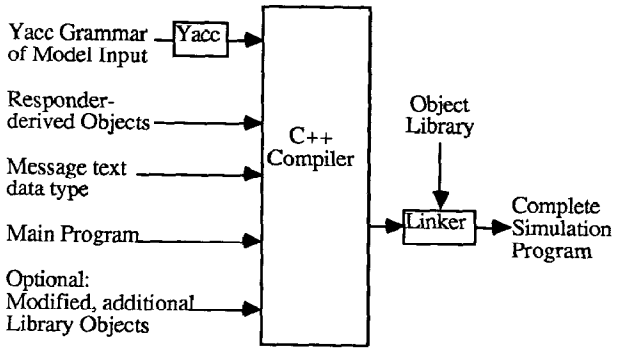

Figure 4: Steps in Creation of a Complete Simulator

To begin with, simulation systems inevitably require an input file that describes the configuration and parameters of the system to be simulated. This file has an arbitrary syntax, and normally requires a non-reusable piece of code to read the file.

To simplify this process, in our system the user first writes a Yacc [13] grammar to describe the syntax of the model input file. The actions associated with the grammar rules instantiate the necessary NODE objects (via `CreateNode`) and other user defined objects. The use of Yacc simplifies maintenance of this portion of code.

The Yacc grammar is then processed by the Yacc program, which generates a C++ compatible parser in the form of a function named yyparse().

The user must also provide:

- RESPONDER-*derived objects:* each provides a `RespondTo` operation to simulate each distinct type of physical process in the system
- *Message text data type*
- *Main program:* calls yyparse() followed by operation `Simulate`
- *Optional objects:* required by RESPONDER-derived objects, objects reimplementing any library objects (e.g., to employ a new parallel simulation algorithm), or additional library objects (e.g., a SOURCE class derived from the SEQUENCER object to introduce new messages into the simulation.)

All of these C++ programs are then compiled and linked with necessary objects from the library to produce a complete simulation program.

Example: The Reed, Maloney, McCredie Benchmark: Reed, Maloney, and McCredie [19] simulated a variety of queueing networks (tandem; general, feed-forward; cyclic; central server; and cluster networks) using the Chandy-Misra algorithm. We implemented a queueing network simulator using OLPS to measure the same benchmark; it requires only 784 lines of code (including comments and blank lines).

The benchmark requires five types of physical processes: sources, forks, servers, merge nodes, and sinks. It is only necessary to write two RESPONDER objects for this problem: SOURCECLASS and SERVERCLASS. The necessary physical processes are instantiated with the following calls:

Source: `CreateNode(Id, Type, Null, SourceClass, DecFreeRouter)`

Fork: `CreateNode(Id, Type, CMFifoQ, Null, ProbRouter)`

Server: `CreateNode(Id, Type, CMFifoQ, ServerClass, DecFreeRouter)`

Merge: `CreateNode(Id, Type, MultiCMFifo, Null, DecFreeRouter)`

Sink: `CreateNode(Id, Type, FifoQ, Null, Null)`

Null represents the absence of a particular object when none is needed. For example, no messages arrive at a source, and therefore no SEQUENCER is required.

Support of Simulation Languages. The discussion above has focused on how to directly create a complete simulation program using the Object Library. This method requires that the user use C++ to program his RESPONDER derived objects.

We recognize that languages for describing simulations — particularly distributed system simulations — is an active research area. In these situations C++ is not a desirable simulation language, and an alternative is possible. A simulation language may be implemented by including calls to operations of the desired library objects, provided that the two languages share the same linker. The Object Library would fill for the simulation language the role that a run-time library plays for a programming language.

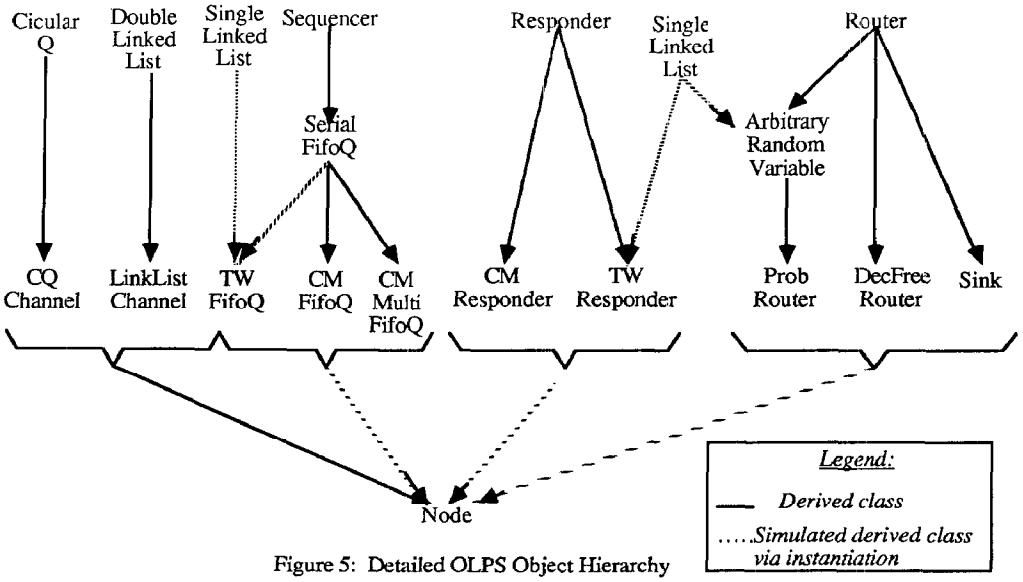

Figure 5: Detailed OLPS Object Hierarchy

2.3 Object Library Components

Figures 1 and 2 present two views of the Object Library. The precise hierarchy is shown in Figure 5. The figure shows that the LINKLISTCHANNEL and CQCHANNEL are not actually base objects; because it maintains a queue, they are derived from a DOUBLELINKEDLIST object or a CIRCULARQ object, respectively. The queue is initialized by the PreLoad operation of §2.1. A ROUTER may base its route on a discrete random variable (object ARBITRARYRANDOMVARIABLE), which in turn is derived from a SINGLELINKEDLIST object.

Objects specific to Chandy-Misra: The CMRESPONDER adds to the user's RESPONDER the automatic generation of null messages in the deadlock-avoidance version of the Chandy-Misra algorithm. The MULTICMFIFOQ is used when there are multiple nodes sending messages to a node to merge the input streams and to calculate the channel time (as defined by Chandy and Misra [5]).

Objects specific for Time-Warp: For the Time-War algorithm, the object library contains a derived object for each SEQUENCER and RESPONDER that can do recovery. Each object derived from the RESPONDER object adds a data structure to checkpoint its state, and adds an operation callable from the SEQUENCER to roll the state back. Each SEQUENCER and derived object adds a data structure to store old input and output messages, respectively. The SEQUENCER maintains the local virtual time for the node and is responsible for initiating the rollback process.

2.4 Instrumentation

OLPS provides two types of instrumentation to collect information on the run-time behavior of a simulation: object measurements and traces. Object measurements are minimally intrusive, while traces may be highly intrusive.

Object measurements. Examples include buffer occupancy, number of rollbacks, waiting times for locks and monitors, and time spent blocked on buffer over and underflow. Each object allocates a private memory area for collecting information on its behavior and collects data as necessary when its member functions are called. Thus statistic collection of objects in different processes do not interfere with each other.

After the simulation completes and the child processes terminates, the single parent process accesses each private data area to output individual object measurements and generate a summary report.

Traces. Traces may be selected which are done independently for each process or collectively for all processes. The second form is highly intrusive because a single lock must be acquired by all processes to log an event. This form is more useful for debugging.

OLPS can produce several types of traces of the runtime simulation behavior, using the LOG object:

1. a trace of when spin locks are acquired and released,
2. a trace of messages that pass through each channel, and
3. user specified events.

OLPS has one further facility. During simulation execution, one can observe imbalances between the rate at which each processor executed events in the following manner. OLPS can, at user specified intervals, display on a monitor during execution a processor number and the number of events executed so far by each processor. This facility can be made arbitrarily unobtrusive to the simulation behavior by selecting a suitably infrequent display interval.

3 PERFORMANCE MEASUREMENTS

In this section we illustrate the use of OLPS in comparing the performance of two simulation algorithms. As a benchmark we use the tandem, cyclic, and central server queueing networks of Reed, Maloney, and McCredie [19]. This benchmark is also used by Wagner, Lazowska, and Bershad [23].

Test environment: Measurements were made on a five processor network on the five processor Firefly, built by Digital Equipment Corporation. DEC describes the machine as follows:

> The Firefly is a closely-coupled multiprocessor machine with 5 processors, 8 megabytes of memory and an Ethernet interface. Each processor is a MicroVAX II processor running at 12 MHZ for an effective performance of about 2 MIPS, or 90 percent of a VAX 11/780 per processor. Each processor has a per-processor cache of 16 kilobytes of memory with the "snoopy cache" (or write-broadcast) protocols for ensuring cache consistency. The caches are direct mapped. A test-and-set instruction is used for synchronization.
>
> The caches are interconnected and connected to main memory by a 10 Mbyte per second memory bus that is separate from the I/O Bus. A cache miss adds 3 or 4 wait cycles to an instruction cycle. We have observed at most a 14 percent degradation in processor performance due to cache misses and bus contention with 5 processors.[9]

Each workstation had sufficient memory to accommodate all executing processes (i.e., no virtual memory or swapping was used). Each workstation ran the V 6.3 operating system. A minimum number of processes other than the simulation was run on the machines during tests (e.g., to manage the display, service network traffic, and provide a command executive). Thus each machine executes a small background load during simulation (e.g., to update the processor clock, reject multi-cast or broadcast packets on the Ethernet).

The time required to create and destroy processes is reflected by the time required to simulate zero jobs in a five node tandem network: 1.21 seconds.

The measurements reported are the means of three to five runs. (Five runs were used when a high variance was observed in the first three runs.) The number of events simulated was chosen so that simulation runs lasted for at least 100 seconds, which from experimentation appeared to be the shortest duration that maximized the speedup. The speedups we report are optimistic because the single processor execution of the parallel simulation performs locking, which results in unecessary context switching.

Tandem network. A tandem queueing network with N processors should achieve near N fold speedup on a workstation with at least N nodes. Ideally, the jobs should pipeline as they pass through the nodes. This behavior was observed by Reed, Maloney, and McCredie [19], subject to bus and memory contention at higher numbers of processors.

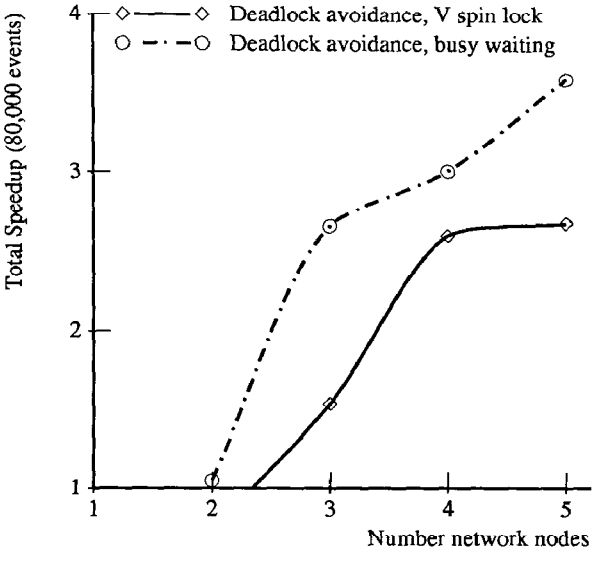

Figure 6: Tandem Queueing Network Speedup

Figure 6 illustrates the measured behavior of two to five nodes with the Chandy-Misra algorithm and eager deadlock avoidance, in the terminology of Wagner, Lazowska, and Bershad. [23]. (Eager avoidance means that a node only sends a null message if its input queue is empty.) We used equal, deterministic service times at each node. Reed, Maloney, and McCredie [19] obtained linear speedup in the two to five processor range on a Sequent Balance 2100. Three factors may account for the limited speedup that we obtained:

1. The five Firefly processors are not homogeneous in the sense that one processor has an additional load from the Q-Bus to which the display and Ethernet interface is attached. This processor is only used in the five node network simulation. Therefore sublinear speedup when the number of processors is increased from four to five is understandable.
2. The bus transfer rate of the Firefly is lower than the 80 MByte per second maximum transfer rates of the Sequent Balance 21000. The queueing network benchmark requires a minimal amount of processing to service an incoming job because a node performs one addition to calculate the job departure time. Thus communication is expensive relative to local processing activity, and accesses to locks occurs frequently.
3. The current version of V on the Firefly serializes accesses to the kernel. Kernel accesses are made in allocating memory and in delaying when using the V spin lock mechanism.

Figure 6 contains two curves reflecting the behavior of two locking policies. One is the V spin lock mechanism. In the event the lock is not available, this mechanism tests the lock once, and then sleeps for one clock tick (10 milliseconds). This cycle is repeated until the lock is acquired. The second lock mechanism busy waits until the lock is available, executing several NO-OP instructions between retries. The lock is tested every 8 microseconds.

When locks are unavailable in the tandem network it is usually because of contention for access to the shared SEQUENCER object between nodes. Since the lock holding time of a process is only a few instructions to enqueue or remove a message, the V lock mechanism with its 10 millisecond delay for not acquiring a lock performs worse than than busy waiting. Busy waiting improves the performance somewhat, but at the same time introduces additional bus traffic due to more frequent lock testing.

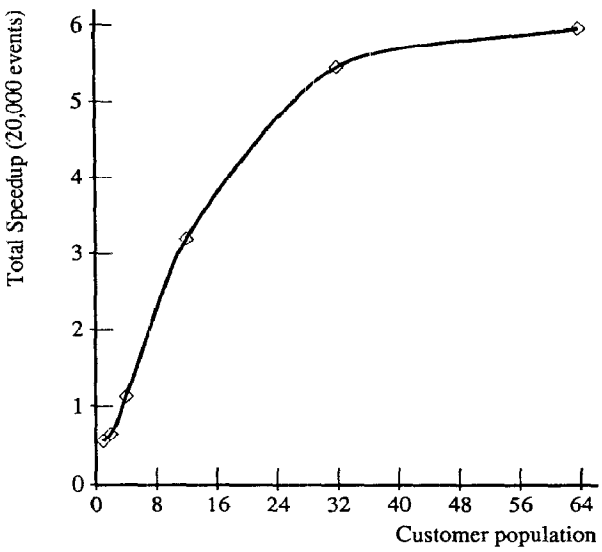

Figure 7: Four Node Cyclic Queueing Network Speedup

Cyclic network: Figure 7 depicts the behavior of our implementation of the Chandy-Misra on a cyclic network of four nodes. Previously Reed, Maloney, and McCredie obtain under two fold speedup. Wagner, Lazowska, and Bershad obtained up to four fold speedup for a four node network. Our speedup exceeds four, which may reflect the fact that the single processor execution acquired locks.

Central Server network: We measured the same central server network (Figure 8) that Reed, Maloney, and McCredie did. The result is illustrated in Figure 9. All results indicate that our Chandy-Misra deadlock avoidance as well as Time-Warp implementations ran slower in parallel than sequentially.

The Time-Warp implementation follows the algorithm described by Jefferson and Sowizral [11]. Ours differs in two respects:

1. "State" in the queueing network model requires only one memory word: the departure time of the last job. Thus we save all states, which eliminates the need to perform a coast-forward step.

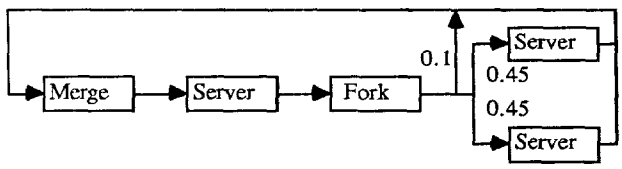

Figure 8: Central Server Queueing Network

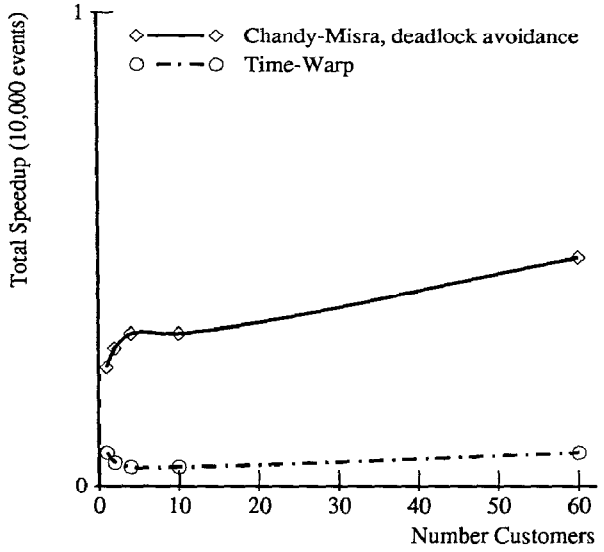

Figure 9: Central Server Queueing Network Speedup

2. Rather than send one antimessage for each outdated message whose effects are being undone in the cancellation step, we send a single message interpreted to mean "annihilate all messages from this source whose time stamp is equal or greater to my time stamp."

Our implementation of the Chandy-Misra algorithm performed two to four times more slowly in parallel than when executed sequentially, which is consistent with Reed, Maloney, and McCredie's results. They obtained a sixteenfold decrease in running time.

Two phenomena may account for our Time-Warp implementation performing more slowly than the Chandy-Misra algorithm:

1. Simulation traces showed that the annihilate messages chase outdated messages around the network. Outdated messages are only destroyed when they are waiting in the input queue of a node and, before the message is processed, an annihilate message arrives. When the number of logical processes in the simulation does not exceed the number of physical processors (as is the case in our implementation), outdated messages are dequeued and processed before the corresponding annihilate message arrives. The number of annihilate messages that originated in our system for 10,000 simulated events ranged from 2198 to 3728.

2. The holding time for locks is longer in our Time-Warp implementation than in our Chandy-Misra implementation. This is because whenever an annihilate message arrives, all messages in the input queue must be searched to delete outdated messages.

4 RELATED WORK

An object orient approach to parallel simulation has also been used by Bezivin [1, 2]. However, our work is novel in two respects:

1. OLPS implements multiple parallel simulation algorithms in one system, allows for new algorithms, and provides a single interface to the simulation programmer.

2. OLPS facilitates comparison of existing algorithms. This is significant, given the limited measured performance results available [19, 12, 20].

Part of the challenge of implementing multiple simulation algorithms on a multiprocessor is to tune the operating system; the parameterized spin locks of §3 are an example. An alternative approach to using an Object Library is to implement a special purpose operating system, as is proposed by Jefferson [12]. The Time-Warp Operating System is a three layer operating system implementing the Time-Warp mechanism. Applications reside in a fourth layer.

So far we have found that an operating system whose structure minimizes the services provided directly in the kernel can be tuned to parallel simulation. In contrast, a special purpose operating system may be less suitable implementation of new parallel simulation algorithms in the future.

Using the terminology of Kaudel [15], OLPS in its present form offers one of four categories of parallel simulation techniques, namely model function partitioning:

- **Independent Replicas:** Run one independent, sequential simulation on each processor. Amenable to Monte-Carlo simulation.
- **Support Function Distribution [3, 7, 24]:** Simulate all events on one processor, but assign support functions (e.g., event list management, random number generation, event logging) to separate processors.
- **Model Function Partitioning [4, 5, 8, 10, 11, 14, 18]:** The simulation events to be executed

are assigned in some manner to multiple processors. This permits simultaneous execution of multiple events by multiple processors.

- **Hybrids:** The above techniques may be combined.

Implementation of the remaining categories within OLPS is feasible, but would require a significant amount of additional work.

5 CONCLUSIONS

Incorporating several parallel simulation algorithms into a library of C++ objects provides several advantages:

- *Modularity:* We can change an object in our library to alter any aspect of simulation, add it to the library, and build a new simulation in a straightforward manner to compare the performance of the original and modified versions.
- *Hierarchy:* The object model forced us to organize the code in a way that identified what was common to all simulation algorithms.
- *Efficiency:* C++ offers the efficiency benefits of C. Only objects implementing the desired simulation technique are linked to the complete simulation program. The object mechanism cost is primarily compile time type checking. C++ also permits in-line code substitution when desired. (The in-line code feature allowed us to replace many Argonne macros [16] with functions, making the resultant code more readable while not sacrificing efficiency.)

The chief difficulty we have encountered is that the present form of C++ does not allow multiple inheritance. Therefore we use instantiation to simulate inheritance, slightly increasing the execution time of the inner loop of the simulation. (Instantiation is evident in the `CreateNode` call, which requires passing pointers to the desired objects which a NODE is constructed from. Pointers slightly decrease run-time efficiency, because calls to inherited operations require an extra level of indirection and cannot be expanded in-line. Unfortunately these calls are in the inner-loop of our simulation (that performs `Input`, then `RespondTo`, then `Output`.)

Appealing directions for further development of OLPS include:

- Creation of a set of *sequential* objects, that would allow construction of a simulator that is purely sequential and not organized as a set of processes. This would permit more realistic speedup estimates.
- Support function distribution techniques,
- Complex simulation stopping rules (Simulation now stops when a specified node executes a specified number of events.),
- Efficient facilities for collection of simulation statistics (as opposed to measurements of the simulation program itself), and
- An interactive user interface to control the simulation.

ACKNOWLEDGEMENTS

This work was sponsored in part by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211. It was also supported by equipment from Digital Equipment Corporation. The author wishes to thank T. Abrams for running the experiments and E. Sznyter for help in formatting the text.

REFERENCES

[1] J. Bezivin. Adapting a Simulation Language to a Distributed Environment. *Proc. 3rd Inter. Conf. on Distributed Computing Systems,* Miami, Florida (Oct. 1982) 596-605.

[2] J. Bezivin. Some experiments in Object-Oriented Simulation. *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA),* Orlando, Florida (Oct. 1987) 394-405.

[3] J. S. Birnbaum. Towards the Domestication of Microelectronics. *Comm. ACM 28,* (1985) 1225-1235.

[4] R. E. Bryant. Simulation of Packet Communication Architecture Computer Systems. Tech Rep. MID,LCS,TR-188, M.I.T., Cambridge, MA (1977).

[5] K. M. Chandy, V. Holmes, and J. Misra. Distributed Simulation of Networks. *Computer Networks 3,* (1979) 105-113.

[6] D. R. Cheriton. The V Distributed System. *Commun. ACM 31,* 3 (March 1988), 314-333.

[7] J. C. Comfort. The Simulation of a Master-Slave Event Set Processor. *Simulation 42,* 3 (March 1984), 117-124.

[8] A. DeCegama. Parallel Processing Simulation of Large Computer Networks. *Sim. of Computer Networks.* IEEE, Colorado Springs (Aug. 1987), 51-62.

[9] Digital Equipment Corporation. *Confidential Information Nondisclosure Agreement No. 458A.* 1988.

[10] B. Groselj and C. Tropper. Pseudosimulation: An Algorithm for Distributed Simulation with Limited Memory. *Int. J. of Parallel Programming 15,* 6 (Oct. 1986), 413-457.

[11] D. Jefferson and H. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism. *Dist. Sim. 1985.* Soc. for Comp. Sim., San Diego (Jan. 85), 63-69.

[12] D. Jefferson. *et al.* Distributed Simulation and the Time Warp Operating System. *Proc. 11th ACM Symp. on Operating System Principles,* Austin, Texas (in *OS REVIEW 21* (5)), (1987) 77-93.

[13] S. C. Johnson. Yacc: Yet Another Compiler-Compiler. *Unix Programmer's Manual Supplementary Documents 1.* Dept. of Electrical Eng. and Comp. Science, University of Calif., Berkeley, CA (1986).

[14] D. W. Jones. Concurrent Simulation: An Alternative Approach to Distributed Simulation. *Proc. 1986 Winter Sim. Conf.* IEEE Press, Wash. D.C., 417-423.

[15] F. J. Kaudel. A Literature Survey on Distributed Discrete Event Simulation. *SIMULETTER 18,* 2, ACM Press, (June 1987) 11-21.

[16] E. Lusk, *et al. Portable Programs for Parallel Processors.* Holt, Rinehart, and Winston, Inc., New York, 1987.

[17] J. Misra. Distributed Discrete-event Simulation. *ACM Computing Surveys 18,* 1 (March 1986), 39-66.

[18] J. K. Peacock, J. W. Wong, and E. G. Manning. Distributed Simulation Using a Network of Processors. *Computer Networks 3,* (1979) 44-56.

[19] D. A. Reed, A. D. Malony, and B. D. McCredie. "Parallel Discrete Event Simulation: A Shared Memory Approach." *IEEE Trans. on Soft. Eng. 14,* 14 (April 1988), 541-553.

[20] H. Sowizral. *The Time Warp Simulation System and Its Performance,* Distributed Systems Research Seminar given at Stanford Univ., 31 March 1988.

[21] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, Reading, MA, 1986.

[22] C. Thacker. The Firefly Multiprocessor Workstation. In *Proc. of the Symp. on Architectural Support for Programming Languages and Operating Systems,* (Palo Alto CA, Oct.). ACM, New York, 164-172.

[23] D. B. Wagner, E. D. Lazowska, and B. N. Bershad. *Techniques for Efficient Shared-Memory Parallel Simulation.* TR 88-04-05, Dept. of Computer Science, University of Washington., 1988.

[24] D. L. Wyatt. Simulation Programming on a Distributed System: A Preprocessor Approach. *Dist. Sim. 1985.* Soc. for Comp. Sim., San Diego (Jan. 85), 32-36.

[25] A. Yonezawa, H. Matsuda, and E. Shibayama. *Discrete Event Simulation Based on an Object Oriented Parallel Computation Model.* T.R. C-64, Tokyo Institute of Technology (Nov. 1984).

AUTHOR'S BIOGRAPHY

MARC ABRAMS received a Ph.D. in Computer Science from the University of Maryland in 1986. He then spent a year at the IBM Zurich Research Laboratory, and is currently a post-doctoral scholar in the Distributed Systems Group at Stanford University. He also developed simulation models for the U.S. Army. His research focuses on the performance of distributed and parallel software.

Marc Abrams
Building 460, room 422
Stanford University
Stanford, CA 94305-2140
(415) 960-0295
marc@pescadero.stanford.edu