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

CP*Sim*: A TOOL FOR CREATING SCALABLE DISCRETE EVENT SIMULATIONS

Bojan Grošelj

BoyanTech, Inc.
7601 Timberly Court
McLean, Virginia 22102, U.S.A.

ABSTRACT

CP*Sim* is a tool that was originally designed for parallel simulations. A strict separation between the CP*Sim* kernel and its application library enabled the creation of a serial version, preserving the user interface in the process. In other words, the same source simulation program written in C that uses the CP*Sim* library may be compiled to run on a personal computer or on a multicomputer. This paper presents the CP*Sim* programming model that enabled scalability and portability. It is also shown that a CP*Sim* simulation executed on a single processor can outperform a classical event-list simulation.

1 INTRODUCTION

Over the past years, the field of parallel discrete event simulation (PDES) has matured and produced several experimental as well as commercially available parallel simulation tools (Baezner, Lomow, and Unger 1994, Nicol 1993, Nicol and Heidelberger 1994, Steinman 1991, Waldorf and Bagrodia 1994). Fujimoto (1990) demonstrated that PDES is well suited for many large discrete event simulations, such as the simulations of communication systems, traffic, computer networks, and computer systems. PDES can handle much larger models and can handle them more efficiently. In the commercial simulation world, the main challenge is to parallelize the existing user-friendly simulation tools. So far, no large commercial vendor has invested time and money to develop a parallel simulation tool with the user interface of a serial simulator. The primary reason is (still) a high cost of parallel hardware, the secondary reason is a somewhat limited modeling capability, and last but not least, developing efficient parallel simulation tools is much riskier than developing fancy user interfaces.

Some commercial vendors of serial simulators are aware, however, that the multiprocessors are becoming increasingly affordable and that the simulations of communication networks can be carried out efficiently in parallel. In order for their clients to switch from serial simulators to their parallel versions, they have to provide the same graphical user interfaces, and preserve the thousands of lines of code already written for serial simulators. Since serial simulators were not "wired" for parallel machines, they can be ported to parallel machines only under certain conditions. Among these conditions is ample parallelism in the model (i.e., computationally intense simulations), access to an underlying language, the ability to schedule future events at known time-stamps, and most importantly, the ability to compute positive lookahead (Nicol and Heidelberger 1994). The positive lookahead for a submodel A means that at certain simulation time t, there exists a guarantee that submodel A may not affect any other submodel until time s, where $s > t$. A good lookahead enables many events to be processed in parallel without any inter-processor synchronization.

There exist other problems when porting serial simulators to parallel platforms. Among them are global snapshots and global variables. Global snapshots mean that at some simulation time, some global information is required. In parallel world, this means that the simulated objects have to be globally synchronized. This problem can be handled by a conservative PDES, using moving window protocols, for example, and utilizing slightly old information (Nicol, Greenberg, and Lubachevsky 1992). Global snapshots are very difficult to implement in optimistic PDES, also known as Time Warp (Jefferson 1985), because of the high cost of global rollbacks. Note that the user's data have to be rolled back as well.

If a parallel simulation tool uses global variables, then any processor should be able to access them. This problem cannot be solved without an additional communication/synchronization cost.

CP*Sim* is a commercially available tool for creating

discrete event simulations, based on a variation of conservative PDES (Grošelj 1994). CP*Sim* was developed primarily as a parallel simulation tool for large high performance discrete event simulations. Therefore, CP*Sim* does not have a graphical user interface, and the user is expected to be fluent in C. As an experiment, CP*Sim* was modified for serial simulations, preserving the portability of the user source code. Surprisingly, the serial version performed better than a classical event-list simulator.

Currently, the CP*Sim* simulation model has to be represented as a directed graph of communicating objects and all variables have to be declared locally in each processor. The granularity of objects is defined by the user. For example, a CP*Sim* object can be a telephone switch or the entire submodel of a superposed serial simulator. (The preliminary studies show that CP*Sim* can be used as a parallel engine for serial simulation tools such as MODSIM[1] or SES-Workbench[2].)

CP*Sim* consists of the CP*Sim* kernel and the CP*Sim* library. The CP*Sim* kernel provides for synchronization, scheduling, deadlock prevention and message passing on multicomputer platforms. The CP*Sim* library consists of C functions that are used to build an application program. These functions enable simulation programmer to

- define the simulation model
- assign simulated objects to a selected number of processors (automatically or manually)
- generate, appoint, and destroy events
- use various pseudorandom number generators
- print simulation reports

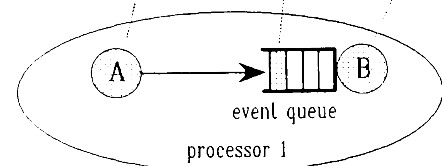
There exist two versions of the CP*Sim* kernel, CP*Sim* 1.0 and CP*Sim* 1.0s. The former is designed for parallel architectures (e.g., iPSC/860[3], CM-5[4], networks), and the latter for uniprocessors (e.g., PCs, workstations). The libraries are the same for both versions. Therefore, the same application program may run on a PC or on a CM-5.

Besides the obvious advantage of portability, namely, that the same program can be executed on different machines without any change of the source code, there is also another advantage. A program (possibly a scaled-down model) can be developed on a PC or a workstation, and fully debugged there using a C debugging tool. After the program is running, it can be scaled-up and executed on a parallel machine. The user can specify the number of processors without recompiling the program. From the user's point of view there is no difference between sequential and parallel programming.

2 THE CP*Sim* PROGRAMMING MODEL

The CP*Sim* simulation model is a directed graph of simulated objects. There is an arc from object A to object B if object A can cause an event at object B. An event is, for example, the arrival of a customer, a signal, or a message. In CP*Sim* 1.0, the input graph is static. In future versions of CP*Sim*, the graph may be modified during the simulation.

The object graph is partitioned among the processors, so that the clusters of objects reside on different processors. The underlying programming model is event message passing. The events are passed between objects along the arcs of the object graph. If two communicating objects reside in the same processor, then the event transmission consists of inserting the event data structure into the appropriate input queue of the recipient.



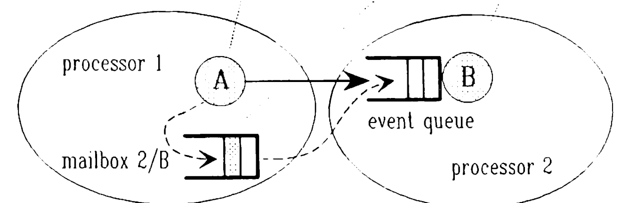


Figure 1: Event Message Passing in CP*Sim*

Besides regular event messages, CP*Sim* employs the so-called null messages. Null messages are synchronization messages that carry time-stamps (or their lower bounds) of future event messages.

Figure 1 shows the CP*Sim* implementation of the event message passing mechanism on a single processor machine and on a parallel processor machine.

[1]MODSIM is a registered trademark of CACI Products Company

[2]SES-workbench is a registered trademark of SES, Inc.

[3]iPSC and i860 are registered trademarks of Intel Corporation.

[4]CM-5 is a trademark of Thinking Machines Corporation.

In this example, object A causes an event at object B. The user has to call function **esend**, where **index_sender** is the index of object A, **receiver** is the CP*Sim* descriptor of object B, and **event** is the pointer to the event data structure. Function **esend** inserts **event** in the appropriate input queue of object B. If object B is in the same processor, then the event is simply appended to the B's queue for the events from A. If object B is located in a different processor than A, then the event is inserted into B's mailbox. Events are delivered periodically. After B's processor receives the event, the event is appended to the B's queue for the events coming from object A.

Event-scheduling is done locally at each processor. CP*Sim* schedules an event of object A for processing at simulation time t only if object A cannot expect any events before simulation time t (conservative rule). Each object maintains a local simulation time (LST). LST is the simulation time at which the object is ready to accept another event. In general, LSTs are not synchronized.

The objects that are allowed to process events according to the conservative rule are *unblocked*, all other objects are *blocked*. The main task of the CP*Sim* kernel is to unblock as many objects as possible. This task is much easier to accomplish in the serial version. The parallel version has to employ an additional protocol for deadlock prevention.

One of the benefits of a conservative PDES is that the time-stamps of events that arrive at object B from another object A are non-decreasing. Therefore, the events can always be inserted at the tail of the queue. This insertion takes constant time. The scheduling (deletion) of an event takes slightly longer. For example, let us assume that object B has three input queues. The next event to be scheduled at B is one of the events at the head of each input queue. In our example, only three events have to be examined. If the event with the smallest time-stamp is a non-null event, then it can be processed immediately. If it is a null event, then object B is blocked and the scheduler has to examine another object. The experiments with CP*Sim* show that on the average about one half of all objects is unblocked. Hence, if the number of input queues at each object is small, then the insertion/deletion of an event takes constant time. This fact explains why CP*Sim* can outperform the event-list simulation with $O(\log N)$ overhead per each event insertion/deletion, where N is the size of the event list. In CP*Sim*, however, there exists additional overhead associated with the procedure that increases the time-stamps of null-messages. The overhead of this procedure is $O(\log n)$ per event, where n is the number of objects located in the processor (Grošelj 1994). Note that in most simulations N is much larger than n.

3 CREATING SIMULATIONS IN CP*Sim*

The process of creating simulations in CP*Sim* can be best illustrated by Figure 2 below. The example is a simulation of message passing on a hypercube multicomputer system (BoyanTech, Inc. 1994).

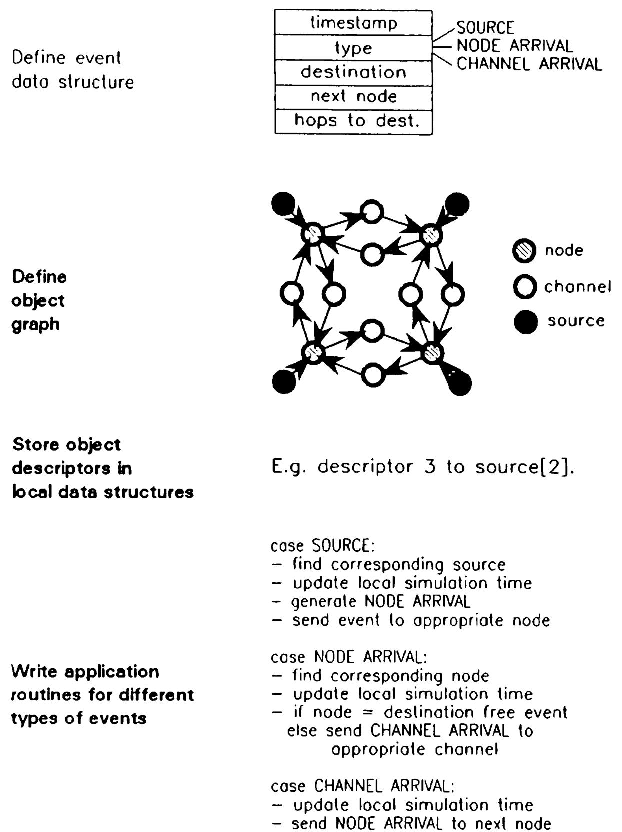


Figure 2: Creating an Application

The model consists of the following objects: *sources*, *nodes*, and *channels*. Each node has a corresponding source that generates messages to random destinations (i.e., other nodes). A node routes an incoming message to one of the outgoing channels following the shortest path from source to destination. Since there are in general several equivalent paths, one is chosen randomly.

The user has to first define the event data structure. Every event must have **timestamp** and **type** fields. In this example, the user might also define fields **destination**, **next_node**, and **hops_to_destination**. Next step consists of defining the object graph. This procedure is accomplished

by calling the functions `new_lp` and `connect_lp` to define new objects, and establish directed communication channels between them, respectively. The user has to store the object descriptors returned by function `new_lp` in user's private data structures. Finally, the user has to write code for processing the particular events. The last step is standard in all event-based simulations. The events in our example are: *source event* (generation of a new message), *node arrival*, and *channel arrival*.

4 PERFORMANCE

The performance of a CP*Sim* simulation depends on many factors, such as hardware, the size and topology of the model, lookahead, pre-sampling of future service times, the allocation of objects to processors, and the rate at which the events are generated in simulation (BoyanTech, Inc. 1994). In general, deterministic CP*Sim* simulations will run fast. A simulation is considered deterministic if service times are constant, they can be generated in advance (pre-sampling), or the lower bounds on service times are larger than 0 (positive lookahead).

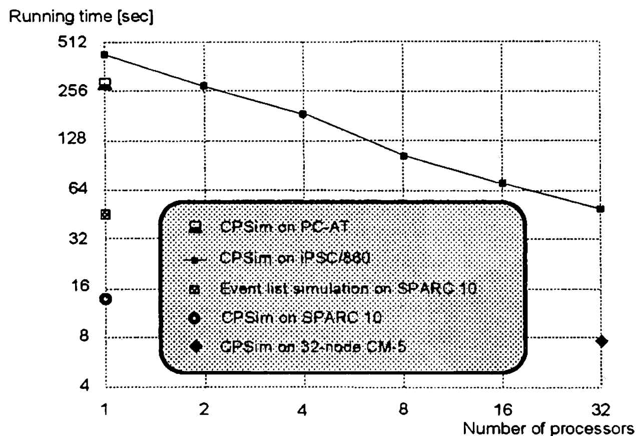


Figure 3: Simulation Performance

Quite commonly, the size of the model is too large to fit in a particular machine. Besides higher overall speed, modern multicomputers provide a substantial amount of memory, so they can handle larger models. Figure 3 shows performance for the CP*Sim* simulation of a 5-dimensional hypercube on four different platforms:

1. IBM/PC-AT[5] with Intel 80386 processor
2. Sun SPARCsystem[6] 10 (sun4m)
3. Intel iPSC/860 with 1, 2, 4, 8, 16, and 32 processors
4. Connection Machine CM-5 with 32 processors

As a comparison, the same example was run on SPARC 10 using a classical event-list simulation with a heap implementation of the event list.

Intel iPSC/860 is an older multicomputer hence it is not surprising that CP*Sim* simulation on SPARC 10 runs approximately three-times faster than on a 32-node iPSC/860. The CP*Sim* simulation on a 32-node CM-5 was the fastest, however, for this small model it was not much faster than on SPARC 10. The serial version of CP*Sim* was about three-times faster than the event-list simulation, both run on SPARC 10. The performances of a PC and a two-node iPSC/860 are about the same.

Multicomputers exhibit better performance for larger models. A simulation of an 11-dimensional hypercube with 26,624 objects was executed on SPARC 10 and on CM-5. 3,172,847 events occurred during the simulation. The running time on CM-5 was 52 seconds while the average running time on SPARC 10 was 375 seconds. The speedup is more than 7, compared to the speedup of 3 for the small model.

5 CONCLUSION

While parallelizing the existing sequential simulations is quite difficult, the reverse is not. Simulation running under CP*Sim* can be efficiently executed on workstations as well as on parallel machines. Scalability and good performance was achieved by carefully designing the CP*Sim*'s data structures, message passing, and synchronization mechanisms. There are some restrictions imposed by parallel simulation modeling. However, the ability to run very large simulations, and run them efficiently, is worth the trouble.

ACKNOWLEDGMENTS

The author thanks David Nicol of the College of William and Mary for pointing out his work (with Philip Heildelberger) on extending parallelism to serial simulators.

REFERENCES

Baezner D., G. Lomow G., and B. Unger. 1994. Parallel simulation environment based on Time Warp.

[5]IBM and AT are registered trademarks of International Business Machines, Inc.

[6]All SPARC trademarks are trademarks or registered trademarks of SPARC International, Inc.

International Journal in Computer Simulation 4: 183–208.

BoyanTech Inc. 1995. *CPSim 1.0 user's guide and reference manual.* McLean, Virginia.

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

Grošelj, B. 1994. Cocktail party simulation. In *Progress in Simulation, Volume 2*, ed. G.W. Zobrist and J.V. Leonard, 151–200, Norwood, NJ: Ablex Publishing Corporation.

Jefferson D. R. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems* 7: 404–425.

Nicol D., A. Greenberg, and B. Lubachevsky. 1992. MIMD parallel simulation of circuit-switched communication networks. *Proceedings of the 1992 Winter Simulation Conference*, Society for Computer Simulation: 629–635.

Nicol D. 1993. The cost of conservative synchronization in parallel discrete event simulations. *Journal of the ACM* 40: 304–333.

Nicol D. and P. Heidelberger. 1994. On extending paralelism to serial simulators. NASA ICASE Report No. 94-95, NASA Langley Research Center.

Steinman J. S. 1991. Speedes: Synchronous parallel environment for emulation and discrete event simulation. In *Advances in Parallel Simulation*, SCS Simulation Series 23: 95–103.

Waldorf J. and R. Bagrodia. 1994. MOOSE: A concurrent object-oriented language for simulation. *International Journal in Computer Simulation* 4: 235–257.

AUTHOR BIOGRAPHY

BOJAN GROŠELJ is the founder of BoyanTech, Inc., McLean, Virginia. Previously he held computer science faculty positions at the University of Southwestern Louisiana, Lafayette, and at the University of Maryland, College Park. He received B.S. and M.S. degrees in electrical engineering from The University of Ljubljana (Slovenia) in 1978 and 1981 respectively. In 1988, he received a Ph.D. degree in computer science from McGill University. His research interests are focused on parallel discrete event simulation, distributed computing, and combinatorial optimization.