Proceedings of the 1987 Winter Simulation Conference
A. Thesen, H. Grant, W. David Kelton (eds.)

TWO IMPLEMENTATIONS OF A CONCURRENT SIMULATION ENVIRONMENT

Carolyn Hughes
Dept. of Computer Science
Texas A & M University
College Station, TX 77843

Usha Chandra
Dept. of Computer Science
Florida State University
Tallahassee, FL 32303

Sallie V. Sheppard
Dept. of Computer Science
Texas A & M University
College Station, TX 77843

ABSTRACT

This paper discusses the design of a concurrent simulation environment hosted on the Ada programming language at the Laboratory for Software Research at Texas A & M University. This environment was first implemented on a VAX 11/750 single processor system and then ported to a Sequent Balance 8000 parallel computer system with ten processors. Two run-time Ada systems were available on the Sequent: one for sequential and one for parallel. This paper reports our experiences in porting the original software to these new environments.

1. INTRODUCTION

The advent of low cost microprocessors has provided a means to overcome the traditional shortcomings associated with simulation such as high memory requirements and execution time. Distributing simulation over multiple processors has long been analyzed as a possible solution to the above problems. However, distribution of the model introduces some unique potential problems such as incorrect forward simulation and deadlocks that do not occur in the real systems.

Multiprocessor systems have been used in simulation in two ways. The traditional approach has concentrated on distributed simulation where machines are networked to achieve parallelism (Chandrasekaran and Sheppard 1987). Message flow between processors is closely related to the topology of the system being simulated. All communication is handled via message passing because there is no shared memory in these systems. A newer approach has been coined "concurrent" simulation (Jones 1986). These systems allow use of shared memory and load-balancing capabilities of multiprocessor architectures. Most of these machines consist of many processors in a tightly-coupled environment and a single shared memory bus, where cache memories are used to reduce bus contention.

The Laboratory for Software Research has implemented a concurrent simulation environment on both single and multiprocessor architectures. Ada was chosen as the implementation language to explore the role of high level language concurrency in implementing distributed discrete simulation. Since Ada provides concurrency at the source level, the multitasking features of the Ada language were employed for distributing the simulation support functions. In this approach the support facilities and the user model are written in Ada. Functions that can be performed in parallel are coded as separate tasks and the distribution of these tasks onto the available processors is the responsibility of the Ada run-time system. Features of the language are available at the source level to be used in handling system synchronization, deadlock protection, and communication between tasks. Since Ada was designed to be portable across architectures, it should allow the software to execute on not just different run-time systems but on architectures having different numbers of processors. The allocation of processors so as to make maximum use of the available hardware is transparent to the user. We see this potential portability across architectures as being one of Ada's biggest advantages. Once this portability ideal is achievable with suitable Ada run-time systems, then Ada software, such as the simulation environment described in this paper, will be portable without modification of system design.

Our simulation environment employs a hierarchical functional decomposition approach by providing the support functions as concurrent program units (Chandrasekaran 1986, Chandrasekaran and Sheppard 1987). In addition, the independent components of a model are also executed in parallel using a parallel algorithm that decentralizes the simulation execution control and handles deadlocks. This parallel algorithm is based on the relaxation technique suggested by Jones and Schwarz (1980) for constructing parallel solutions to general problems.

The concurrent simulation environment includes a set of simulation primitives that are provided as extensions to the host language, support facilities such as random deviate generators, statistics collection routines and queue handlers, and a control unit that incorporates the above mentioned parallel algorithm. The following sections provide an overview of the system, the relaxation algorithm, and the implementation of the support and decentralized control environments. The final sections discuss and compare the single and multiple processor implementation and present the problems encountered in porting the concurrent simulation environment from the single processor architecture to the parallel system. The paper concludes with suggestions for future work.

2. SIMULATION ENVIRONMENT DESIGN

2.1. Overview

The simulation model is viewed as a set of cooperating processes which communicate with each other through time encoded entities. A set of simulation primitives is provided as an extension to the Ada programming language for the simulationist to use in constructing the model. The environment includes a preprocessor which converts the extensions into compilable code in the host language, namely, the Ada programming language. A software library is provided which includes the control and support modules. The relationship between the various components of this environment is depicted in Figure 1.

The software library shown across the top of the figure consists of concurrent programming units (i.e. Ada tasks) for the various support functions and the generic control module. These are available for inclusion in concurrent simulation systems constructed in the environment.

The bottom portion of Figure 1 shows the steps in model construction and execution using the environment. The simulationist provides the model in Ada extended with the simulation primitives supported by the environment. A model syntactically consists of a main program to hold the common declarations such as the attributes of the entities and simulation termination time, and the code for n user-defined processes. Each of the processes in the user-defined model corresponds to a functionally independent component of the system being modeled and is transformed into an Ada task by the preprocessor. The model with the extensions is processed by the preprocessor to transform it into compilable code in the host language. In addition, appropriate support routines are brought in from the software library and the generic control task is instantiated for each user-defined process. The model is then compiled by an Ada compiler and linked with the appropriate software libraries of the Ada language to produce an executable module. Executing the model after the linking process produces statistical data on the behavior of the real system and other user-defined outputs from the simulation.

2.2. Relaxation Algorithm

An algorithm for concurrent execution of a simulation model is included in the environment which is based on a technique for constructing parallel solutions to general problems. Jones and Schwarz (1980) define three possible techniques for constructing parallel solutions. The hierarchical functional approach identifies the independent parts of the problem and executes them in parallel. The divide and conquer approach partitions the data, replicates the processes and performs the same operations simultaneously on the subsets of the data. The relaxation approach uses asynchronous processing as the basic principle. Of the above three techniques, only the relaxation technique processes the data produced by other processes asynchronously. Though the processes depend on data produced by other processes, iterations within a process need not be synchronized with the operations of the other processes. The relaxation technique is comparable to the process interaction strategy. While the data flow of the relaxation approach is similar to the entity or object flow of the process interaction strategy, the major difference is that time stamps are associated with the entities or objects to model the time-phased interaction of the real world processes. Further the notion of complete asynchronous processing suggested by Jones and Schwarz (1980) must be relaxed to allow proper and correct forward simulation. In a concurrent simulation system a process can proceed asynchronously only as long as the correctness of simulation is assured. However, deadlocks can be caused by processes waiting on each other to send entities or messages. The parallel algorithm for simulation allows processes to proceed in parallel as long as no incorrectness is introduced; then it blocks the corresponding process from proceeding further, lets the blocked process communicate with its predecessor processes to resolve such blockings and detects and avoids deadlocks. The complete details of the relaxation algorithm adapted for use in simulation are given in Chandrasekaran (1986).

2.3. Support and Control Environments

All the functionally independent units of the simulation environment such as the random variate generators and statistics collection and queue handling routines have been implemented as concurrent programming units to comply with the goal of developing a truly distributed simulation environment.

The simulation system has to provide a means of generating random variates to model stochastic real systems. The simulation environment provides eight streams of uniform (0,1) generators and random variate generators for exponential, normal, Poisson and uniform distributions (Chandrasekaran and Sheppard 1986).

The concurrent simulation system provides automatic statistics collection and reporting facilities. Statistics is collected on two primary objects of a model: the processes and the entities. The parameters of interest for data collection for a process are process utilization and process idle time in percentage, message buffer length and number of occurrences of deadlock. Parameters of interest for an entity are the wait time, service time at each process and the total time spent by an entity at each process and in the system. The average, variance, standard deviation, minimum, maximum and the number of observations are reported for each data item.

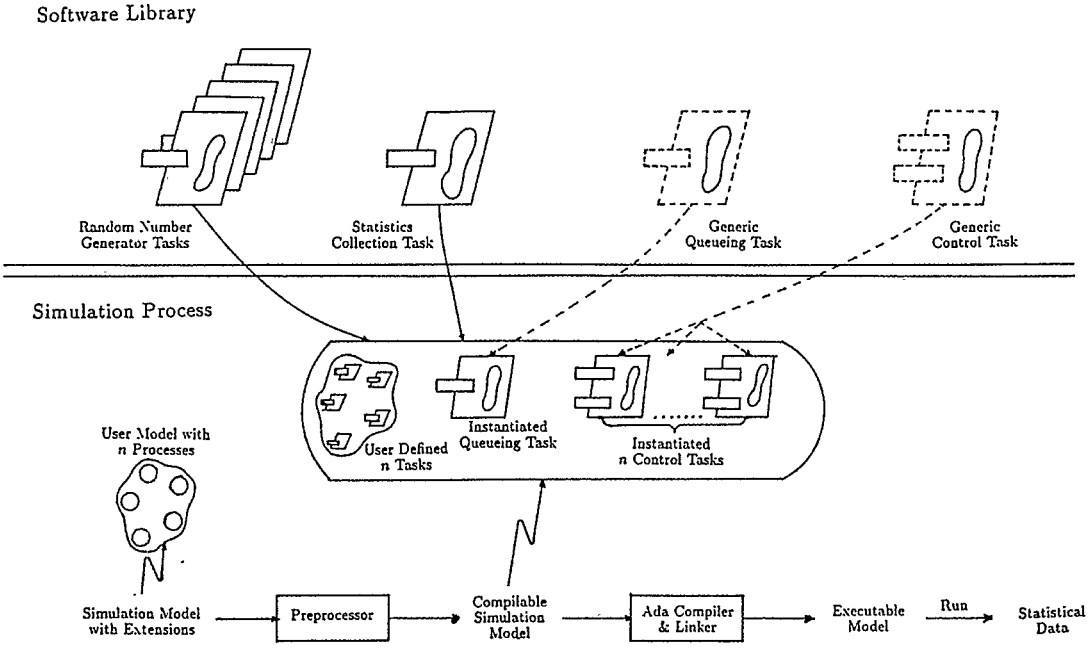

FIGURE 1

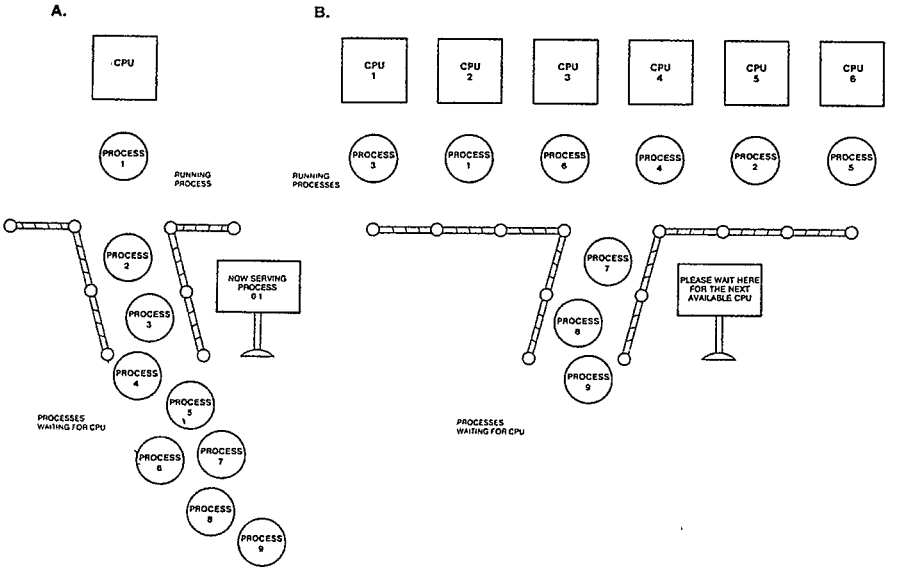

FIGURE 2

Essentially a process receives entities from other processes in the form of messages and enqueues them on a FIFO basis to ensure correct forward simulation. Further, it also removes the enqueued entities one-by-one, does the user-specified operations on the entities and either terminates the entities or sends them to another process. The queueing facility has been provided as a generic package that supplies a template for defining the structure of the queue and the operations to be performed on the queue.

The control environment consists of a generic program unit that utilizes the distributed simulation algorithm developed and is responsible for the progress of the simulation. The extended relaxation algorithm has been abstracted as a task type of the Ada programming language and a control task of this type is instantiated for each user-defined process in the simulation model.

3. SINGLE AND MULTIPROCESSOR IMPLEMENTATIONS

3.1. Overview

The original implementation of the extended relaxation algorithm for distributed simulation emulates parallelism via multitasking on a VAX 11/750 single processor system using the Digital Equipment Corporation (DEC) Ada compiler. In an effort to achieve true parallelism in the concurrent simulation environment, the environment has been ported to a Sequent Balance 8000 employing ten processors and using the Verdix Ada compiler. Two run-time systems are available on the Sequent: sequential and parallel. Figure 2 summarizes the uni- and multi- processor systems. In this example, there are nine executable processes, but the uniprocessor system in Figure 2-A can execute only one at a time. The six-CPU Balance system illustrated in Figure 2-B can execute six processes simultaneously, so each process spends less time waiting for a CPU.

The Sequent Balance 8000 parallel computer system consists of ten general-purpose 32-bit processors in a tightly-coupled environment and a single shared memory bus, where cache memories are used to reduce bus contention. The system is expandable up to twelve processors. This system allows use of shared memory to enhance resource sharing and communication among different processes and provides load-balancing capabilities. The CPU's automatically schedule themselves to ensure that all CPU's are kept busy as long as there are executable processes available. Each process/task is assigned to a processor but at any time may be suspended, swapped or switched to another processor for any of several reasons including waiting for a rendezvous or input/output. All task assignment and scheduling is under the control of the operating system which is also distributed. At any one time there are n-1 processors available for use since the operating system must reside on at least one processor though it can be distributed over more than one if processors are available. (Balance Technical Summary 1986)

The goal of porting the concurrent simulation environment was two-fold. The first goal was to duplicate previous results in sequential mode on the parallel machine. The second was to duplicate these same results in parallel mode. The first goal was achieved with only minor roadblocks, while the second goal has been difficult to attain as discussed below.

3.2. Portability

Minor changes were needed to successfully port between DEC Ada and Verdix Ada. The Verdix Ada compiler was much more careful in its passes citing several potential problems not detected by the DEC compiler. One unreachable exit was found and several warnings were produced for possible non-valued variables. The Verdix compiler had difficulty with arrays of character and these were changed to string types. It is not clear whether this was caused by an idiosyncracy of the program or a fault in the compiler. Other changes were basically file input/output oriented and were due to the Sequent operating system, Dynix, which is Unix-based, whereas the VAX runs the VMS operating system. File names and their extensions had to be edited. And, while the VAX did not distinguish between upper and lower case letters, the Sequent was case sensitive.

Perhaps the biggest problem was user portability between two diverse operating systems. Converting from one system to another was not easy because of radical differences in command structure for compiling, linking and executing Ada. The VAX expected a full sequence of commands each time a compile was warranted. The Sequent required one simple command for a recompile and linking was done automatically.

3.3. Status of Implementations

The above changes were sufficient to ensure successful compilation. The sequential run-time system on the Sequent duplicated the results from the VAX 11/750. So, without too many problems, Ada was easily ported to the Sequent.

The parallel implementation has proven to be much more challenging. Several different problems were identified and some of these were resolved. The original simulation environment included output statements in each of the tasks. As the tasks were instantiated and executed, the output from the various processors became intermingled. Processes were writing at the same time and interfering with each other. This problem was first solved by locking the output statements thereby assuring sequential execution. Later, a single task was instantiated to handle all output.

Although the Ada code compiles successfully, the simulation deadlocks and has to be aborted. It is not clear whether this is caused by the simulation environment or by the run-time environment. The Sequent Ada run-time environment essentially only allows one task to be run at a time. All variables are placed in shared memory and are locked from other tasks when a task is in the run-time environment. Therefore, the parallel run-time environment is essentially sequential. For some reason, as a task enters the run-time environment and locks the shared memory, it is not unlocking shared memory when it exits. Therefore, no other tasks can enter and continue processing and the program deadlocks. As of this writing, no solution has been found to this problem.

The biggest disadvantage in working in parallel environments is the lack of effective debugging tools. While there is a parallel debugger available on the Sequent for the parallel C language, this debugger has not been made available for Ada. Old methods were employed such as extra print statements, commenting out sections of code, etc. It was possible to abort the various tasks and get a core dump of each one which could be analyzed to see what the process was doing at the time of the abort. Unfortunately, the only thing discovered was that all tasks were deadlocked.

We currently have the Beta release of the Ada run-time system and are working with Sequent to resolve our run-time problems. Status of correcting these problems will be reported on at the conference.

In our experience, the state of the art of parallel run-time system environments has not matured to offer a truly parallel system. Yes, small programs can and will run in parallel, but a substantial size program can halt all processors leaving the user without a clue as to the problem. Parallel debugging tools are also not yet sufficiently advanced to help the programmer. Additional research in these areas is required.

4. FUTURE WORK

In general, the concurrent simulation environment appears to be feasible. There are certainly other enhancements that need to be done. One improvement would be to take advantage of running on a shared memory system such as the Sequent instead of a distributed system with no shared memory. Fine-tuning the simulation environment in this way would optimize execution time on the Sequent. Our future plans include prototyping an interface to the concurrent simulation environment which will automatically partition the simulation model into processes which can be run in parallel (Hughes 1987). Of further interest are studies comparing the sequential execution with the parallel execution of simulations.

REFERENCES

Balance Technical Summary. Sequent Computer Systems, Inc., Man-0110-00. Portland, Oregon, 1986.

Chandrasekaran, U. (1986). The design and implementation of a distributed concurrent simulation environment. Ph.D. Dissertation, Texas A & M University, College Station, Texas.

Chandrasekaran, U. and Sheppard, S. (1986). Implementation and analysis of random variate generators in Ada. Journal of Pascal, Ada and Modula-2, 4, 5, 27-39.

Chandrasekaran, U. and Sheppard, S. (1987). Discrete event distributed simulation. In: Proceedings of the Conference on Methodology and Validation. Orlando, Florida, 32-37.

Chandrasekaran, U. and Sheppard, S. (1987). Discrete event distributed simulation-- a state of the art survey. Technical Report 87-005, Department of Computer Science, Texas A & M University, College Station, Texas.

Hughes, C. (1987). Automatic partitioning for parallel simulations via an intelligent concurrency configurator. Ph.D. Research Proposal, Texas A & M University, College Station, Texas.

Jones, A. and Schwarz, P. (1980). Experience using multiprocessor systems-- a status report. Computing Surveys, 12, 2, 121-165.

Jones, D. (1986). Concurrent simulation: an alternative to distributed simulation. In: Proceedings of the Winter Simulation Conference. Washington, D. C., 417-423.

AUTHORS' BIOGRAPHIES

CAROLYN HUGHES is a Ph.D. student in the Department of Computer Science at Texas A & M University. She was formerly the Director for Computer Activities in the College of Technology at the University of Houston. Her research interests include parallel processing, simulation and real-time systems.

Carolyn Hughes
Dept. of Computer Science
Texas A & M University
College Station, Texas 77843
(409) 845-0299

USHA CHANDRA is a visiting assistant professor of Computer Science and Electrical Engineering at Florida State University. She received her Ph.D. in Computer Science from Texas A & M University in August 1986. Her current research interests include parallel/distributed processing, simulation and artificial intelligence. She is a member of ACM and IEEE societies.

Usha Chandra
201 Love Building
Dept. of Computer Science
Florida State University
Tallahassee, Florida 32306
(904) 644-4062

SALLIE V. SHEPPARD is presently professor of Computer Science and Director of the Laboratory for Software Research at Texas A & M University. She received her Ph.D. in Computer Science in 1977. Her research interests include simulation and AI support for software engineering. She was a Halliburton Professor in 1983 and received the Texas A & M Former Students Outstanding Teaching Award in 1985.

Sallie V. Sheppard
Dept. of Computer Science
Texas A & M University
College Station, Texas 77843
(409) 845-5466