Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

SIMULATION OF QUEUING SYSTEMS ON A HIGHLY PARALLEL COMPUTER SYSTEM

Isis D. Rosello
School of Computer Science
Florida International University
Miami, Florida 33199

John Craig Comfort
School of Computer Science
Florida International University
Miami, Florida 33199
e-mail Comfort@SERVAX.Bitnet

ABSTRACT

The purpose of this research was to implement the simulation of specific queuing systems on the Encore Multimax computer system. The environment partitioning approach to distributed simulation was used to implement parallel simulations of three benchmark queuing models to show performance improvement. The execution times obtained for the queuing systems were better with environment partitioning than without. However, this only occurred when the objects to implement the simulation were tightly coupled and were not hiding a lot of information. Object oriented techniques were not completely applicable because the time spent in message passing between processors was large compared to the time spent in a processor's local computation. While speedups of up to 2.0 were obtained, this is far smaller than prior simulation studies have suggested are possible for a computer with faster inter-processor communication. Thus, the Multimax was not a suitable machine for distributed simulation employing object oriented design and environment partitioning.

1. INTRODUCTION

There are two primary approaches to distributing discrete event simulation tasks over the set of available processors. In the first approach, **model partitioning**, the model is divided into components or subsystems, which are then assigned to physical processors, and **Environment Partitioning**, in which those CPU-intensive functions required to support the simulation are identified and assigned to processors ancillary to the main simulation processor. The tasks requiring the most computer time in many simulations, especially in queuing systems, are priority queue processing, statistical accumulation, and random number generation. These objects required to perform the simulation are identified, constructed, and then assigned to independent processors.

Model partitioning can be successful in applications that permit partitioning with a high level of computational granularity (the ratio of time spent in a processor's local computation to the time spent in message passing between processors). Simulations of queuing systems have low granularity since they perform little computation per system state change. Since environment partitioning is appropriate for systems of low computational granularity, this approach can be used for the simulation of queuing systems.

1.1. Object Oriented Programming and Simulation

Object oriented programming is a promising strategy for obtaining both model flexibility and ease-of-use in simulation software. The object-oriented paradigm is a design and programming discipline that focuses on the objects (distinguishable components of the system) that make up the system rather than on the overall function of the system. The modeling task consists of creating, manipulating, and destroying objects using established methods to emulate the operation of the real system.

Object-oriented programming techniques can be used to develop the queuing system simulation software. Environment partitioning is easily expressed, since identifying computation intensive is fully supported by the object-oriented paradigm.

1.1.1. Principal Objects in Queuing Systems.

As mentioned earlier, the tasks requiring the most computer time in queuing systems are priority queue processing, statistical accumulation, and random number generation. The objects performing these tasks and their activation points, are:

PRIORITY QUEUE PROCESSOR

(instances: event set, wait queue, idle queue)

Initialize

Schedule(identifier, priority)

Create a notice containing the identifi er and its specified priority. Schedule this notice in order of increasing priority.

Next (identifier, priority)

Extract the notice with lowest priority from the queue.

Terminate

RANDOM NUMBER GENERATOR

(instances: inter-arrival times, service times)

Initialize (type, seed, mean)

Accept the type of the streams to be gener ated (negative exponential, uniform, or constant), its initial seed, and its expect ed mean, then initiate the process.

Generate (value)

Return the next value in the specified stream.

Terminate

STATISTICAL ACCUMULATOR

Initialize

Set all entity times of creation and times of change to zero. Set all the facility times of creation and time of last change to zero.

Accrue (entity, state, time)

Accumulate the sums and sums of squares necessary to compute first and second order time and utilization statistics.

Aggregate (accumulated statistics)

Using the raw statistical information accu mulated by Accrue, compute the aggregated second order time and utilization statis tics, then transmit them to the requesting process.

Reset

Reset the accumulators for time and utiliza tion statistics.

Terminate

2. STATEMENT OF THE PROBLEM

Programs to simulate discrete systems tend to consume a lot of computer time. It seems reasonable to attempt to execute these programs on super computers, or on distributed multi-processor systems. The published results of these attempts have been disappointing, especially for an important subclass of DES models involving queuing systems. It is not even known how much performance improvement is possible in the simulation of a general queuing system. Hence, by implementing the simulation of specific queuing systems, performance improvement, if any, can be shown. This study consisted in employing environment partitioning and the object oriented strategy to implement the simulation of the three queuing systems (**MMc**, **CR5**, and **CQn**) defined below.

Use of MMc, a single level multi-server queue (Figure 1) allowed the investigation of a system under different object loadings. The relative object loadings were varied by varying the size of the event set. This model was included in the study because it is typical of the queues that appear in real simulations.

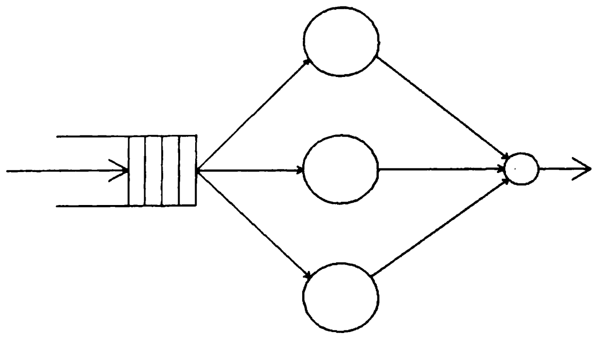

Figure 1. A c Server Queue (MMc)

CR5, a five node central server model (Figure 2) is a highly simplified model of a computer system containing a CPU and two I/O devices. This model was included in the study because it has been extensively analyzed and simulated. This model was originally introduced by D. A Reed. The presence of the fork and merge nodes is an artifact of the specification language he used. Since the fork and merge nodes require zero service time, they were eliminated from the implementation.

CQn, closed circular queuing network containing n queues (Figure 3) was included in the study because it should permit easy model scalability. It should be possible to maintain a relatively constant efficiency by employing **k*n** processors as **n** increases.

The results of the implementation of the queuing systems simulations were compared to the results obtained by Comfort [Comfort 1991] where these queuing models were only simulated, not implemented. Since the implementation of the target queuing models was not studied before, only experimentation could show if improvements could be obtained.

3. THE APPROACH

The computer system for running the distributed simulation was the **Encore Multimax** running the UMAX 5 operating system, which is based on UNIX System V (release 2.4.1). The Multimax has 80 Mbytes

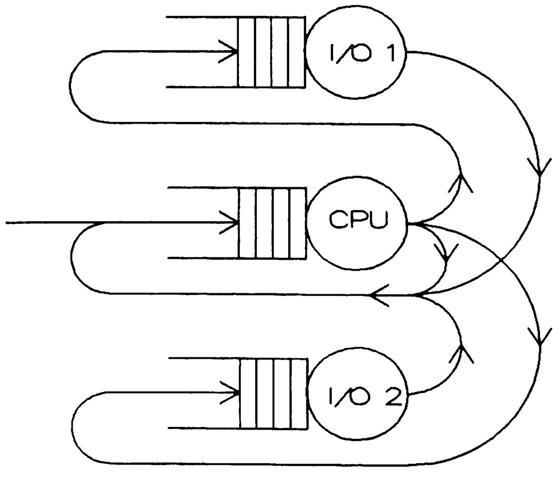

Figure 2. The CR5 Queuing System

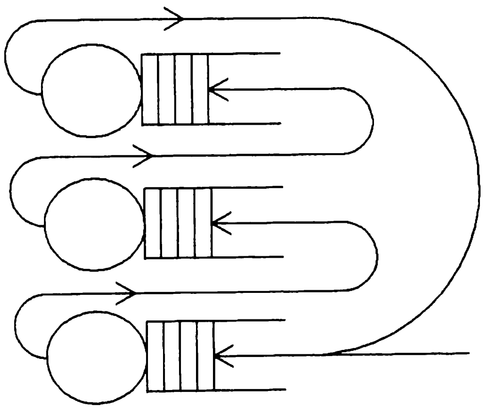

Figure 3. An n Station Circular Queuing Network (CQn)

of memory shared by 8 NS32532 processors. Each processor, running at 8 MIPS, has sufficient memory to contain the objects assigned to it. One of these processors, the master, contained the controlling simulation program. The other processors, called the ancillae, contained instances of objects (priority queue processing, random number generator, statistics) needed in the simulation.

The processors are linked via a high speed bus providing communication between the master and the ancillae. Only the master processor initiated communication. Since the priority queue processing, statistical accumulator, and random number generator behaved somewhat independently, there is the potential for parallel execution. The speed of simula tion was accelerated by exploiting such parallelism. The Multimax, a highly parallel computer system, was used to implement the programs developed.

The implementation language chosen is C++, an object oriented language that was used to implement the queuing models. The C++ language was chosen for the implementation because it has fast ex ution times and is highly portable. C++ code is preprocessed to give C object code, which can then be linked and executed like ordinary C programs. The C++ compiler used was AT&T C++ 2.1.

3.1. Object/Instance Partitioning

The simulation objects were assigned to the ancillae in various ways. An obvious way was to assign the controlling simulation program to the master and each object to one ancilla. The gang scheduling operations supplied by the operating system were used for the processes assignments to processors. A gang is a set of processes where the number of processes that can enter the gang is limited by the number of CPUs within the gang. Since the Multimax has only seven processors available for processor scheduling operations, a maximum of seven processes could be assigned to independent processors. The processes were permanently assigned to a specific processor within the gang. A process was assigned to the gang if the gang was valid; otherwise, it ran on any processor available. The current operating system did not allow assigning more than one process to a specific CPU. This decreased the number of experiments which could be performed. Different assignments of object instances to processors were tested in order to experiment with environment partitioning, concurrency, and the Multimax.

The main objective of this research is to evaluate the performance of programs simulating the queuing models CR5, CQn, and MMc using the Multimax. The actual implementation of the queuing models is what makes this study unique. In previous work, these queuing models have only been simulated, where the simulation supporting objects were simulated instead of implemented. In this study, parametric information about each supporting object was produced. This information included the object execution times (execution time required by each critical function performed by the queuing program), invocation time (time required for call/return), and the interface time (time required to transfer information to/from the ancillary). The generated parametric information was a dominant factor in the implementation of the supporting objects because it gave insight about the simulation objects running on the Multimax.

In order to determine the effectiveness (or lack thereof) of a particular partitioning scheme, two measures were used: speedup and relative efficiency. Speedup is defined as the ratio of the original (uniprocessor) run time divided by the multiprocessor run time. Relative efficiency is the speedup divided by the number of processors. For example, if the original simulation required 10 seconds to run one block, a two processor system required 6 seconds, and a three processor system required 5, the speedup would be 1.67 and 2.00 respectively, while the efficiencies would be 0.84 and 0.67.

The main steps for this study were:

1) Producing parametric information about each simulation supporting object:

In order to compute execution time, separate programs were written to invoke the critical functions of each queuing model. Confidence intervals for the execution time were computed, after removing the effects of the controlling software. Estimates about invocation time were obtained by writing other programs that per formed the critical operations repeatedly. Interface time was computed by writing separate programs to time inter-processor information passing.

2) Implementing the simulation of the queuing systems:

Programs specific to each queuing system were written using the simulation support objects. The queuing system programs were run using parameters such as number of servers, seeds for the random number generators, number of blocks, block length, etc. Actual run times were ob served for each queuing model.

3) Experimenting with assignments of processes to processors:

Different assignments of simulation supporting object instances to processors were made to determine the best run times that could be obtained for the queuing models. Then the total speedup was computed to demonstrate if the Multimax was suitable for distributed simula tion.

4. RESULTS

4.1. Parametric Information

Separate programs were written to call repeatedly procedures and functions with a different number of parameters to measure their invocation time. The procedures and functions had the same number of parameters as the RNG, PRQ, and STAT's critical functions. The critical functions were GetRandom, Schedule, and Change, respectively. Therefore, the invocation time for the critical functions could be computed and not counted in their execution time. After calling the critical functions template 10,000 times for ten batches, the invocation time was consistently 2.5 microseconds.

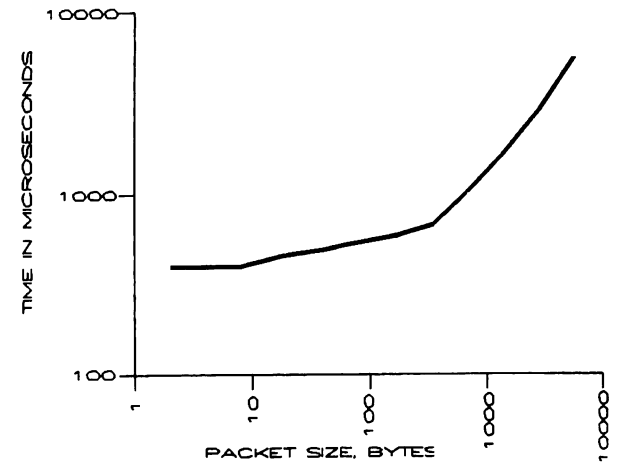

Figure 4. Transmission Time per Byte/ No Response Required

Separate programs were written to compute the time required to transfer information to/from the ancilla. A program was created to time the transmission of bytes between two processes, each assigned to a different processor. Figure 4 illustrates the time in microseconds that was spent in transmitting bytes from the sending process to the receiving process. The mean transmission time obtained was approximately 0.93 microseconds per byte.

Another program was run to time the transmission of bytes but a response was sent from the receiving process to the sending process. This behavior was needed when the sending process' functionality depended on the received response. Figure 5 illustrates how time increased depending on the number of bytes transmitted. In this case, the mean transmission time was 2.35 microseconds per byte. Obviously, a large overhead was introduced when an answer was required

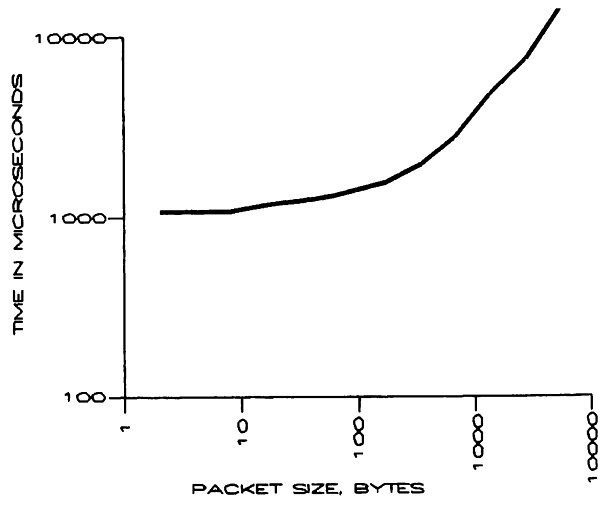

Figure 5. Transmission Time per Byte/ Response Required

from the receiving process. The transmission of a small number of bytes was not considered worthwhile because of the slow message passing mechanism.

The size of interface time was a dominant factor in the implementation of the objects used for thesimulation, since it will be present in all communication between processes residing on different processors. It was not worthwhile to transmit small number of bytes, because the interface was too expensive. This was definitely a very important factor in the design of the simulation supporting objects.

Each program used to compute execution times for the simulation objects ran on one processor. The objects were not created by a particular process running on a different processor. There was no need for communication between processes, since environment partitioning was not implemented.

A program was written to compute the execution time for the critical function **GetRandom** of the Random Number Generator (RNG). It took 82 microseconds to compute a random number when using a negative exponential distribution. Thirty microseconds were spent when the stream to be generated was uniform.

For the Statistical Accumulator (STAT), it took 80 microseconds to perform the critical function **Change** used to change from and old to a new state in the simulation.

Finally, for the Priority Queue (PRQ), it took 10 microseconds to perform the critical function **Schedule** used to create a notice containing the identifier and its specified priority.

The execution times were much smaller than the interface times. Clearly, careful design of the simulation support objects was necessary to obtain advantages from using environment partitioning in implementing the

4.2. The Simulation Support Objects

In the following, the term **Simulation Support Object** will be used to name objects whose only purpose is the facilitation of communication between a program running on one processor and an object residing on another. These pairs of objects, as shown schematically in Figure 6, will serve as the software interface between these communicating entities. To ease the task of the programmer using this system, the supporting objects will have exactly the same interface (calling sequence, etc.) that the original objects employed. If the support objects perform actions no more sophisticated than message passing, they are called **trivial**.

4.2.1. Random Number Generator

As described above the controlling program created the

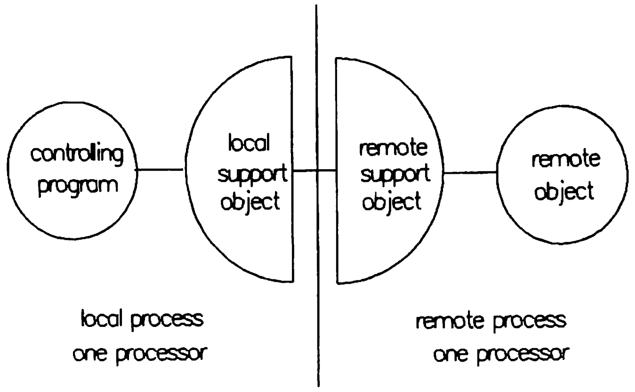

Figure 6. Interface between the Controlling Program and a Remote Object

local support RNG object which had the same interface as the actual RNG object. On another processor was created the remote support object, which waited for messages from the local support object, and called the RNG to get random numbers. The remote process passed the obtained random number to the local process, to be returned to the main program. The system call Fork was used to create the waiting process. Communication between the support objects was through messages, using the UMAX message operations **msgrcv** and **msgsnd** to receive and send messages, respectively.

A test program was written to compute the execution time for the critical function GetRandom. The main program created the local RNG object (instead of the actual RNG object) to explore the effects of environment partitioning. The controlling program was assigned to a processor, and the RNG to another. It did not matter to which processors the processes were assigned, since all the processors were identical. The execution time increased significantly (from 82 to 1140 microseconds) when the stream type was negative exponential, and from 30 to 1170 when the stream type was uniform. This behavior verified that much time was spent in inter-process communication. Therefore, it was not efficient to receive a random number from the remote support RNG, since every time the GetRandom method from the local process was called, as the time to transfer the random number was much greater than the time to compute it.

A change was necessary to get random number generation to be at least as fast as when environment partitioning was not implemented. The local supporting RNG was modified to keep a batch of random numbers. This object only communicated with the remote support RNG to receive a new batch of numbers when its current batch was empty. Instead of the waiting process having only one value ready to return when it received a message from the local support RNG, it had an entire batch of numbers ready. The interface time per random number was greatly minimized. Since a batch of numbers was transmitted instead of one at a time, less communication was needed between the local and remote support RNG's. Some parallelism was exploited because, while the trivial object was storing the numbers and returning them to the controlling program, the waiting process was obtaining the next batch of numbers to be sent.

Experimentation was necessary to determine the optimal batch size for number transmission. By keeping 40 random numbers in the local object, the execution time per number generated was minimized from 1140 to 100 microseconds for negative exponential, and from 1170 to 44 for uniform. There was not an advantage in using environment partitioning because the interface time was larger than the execution time. The interface time could not be minimized any more, because a random number must be returned to the controlling program.

4.2.2. Statistical Accumulator

A similar operation was performed using the STAT object. Since the inter-process communication was slow compared with the execution time for the Change operation, the design did not include a one to one communication between the supporting processes for every statistical operation. The local object did not send a message to perform a statistical operation to the waiting process for every method called, rather, it saved a number of calls with their corresponding incoming parameters. When a set number of calls was reached, it sent the saved calls to the remote process. This approach could be implemented because none of the statistical operations returned a value. While the remote process was forwarding the requested operations on the STAT object, the local process had returned control to its super-ordinate, to allow it continue executing. The STAT object, like the RNG object, could be assigned to a different processor or it could run in the same CPU as the controlling simulation program.

The number of calls to be saved in the local STAT was modifiable. Experimentation when running the queuing models was used to determine the optimal number of calls to be transmitted. In contrast with the RNG, the interface time could be minimized, since the statistical operations did not return any value. The controlling program requested the statistics values after executing each simulation block. The only value that was needed to run the simulation was the state in which the statistics object was in. Therefore, the local STAT needed to keep track of the state value to avoid extra communication with the remote support STAT. This knowledge about the STAT information and functionality was violating the information hiding and loosely coupling principles of the object oriented paradigm. However, it was absolutely necessary to minimize the run times for the queuing models.

4.2.3. Priority Queue Processor

Similarly, the Priority Queue Processor was dealt with. There was not a one to one communication between the local and remote processes for every operation, because the inter-process communication was expensive. The local PRQ support was the most complicated of the supporting simulation objects. The controlling program needed to know the next element that was scheduled with the lower time for the simulation. This required to return the entry identification and time of the scheduled element. However, unlike in the local RNG, the local PRQ could not keep a batch of entities ready to return, because it did not know what entries would be scheduled. The local PRQ therefore needed to know information about the PRQ to minimize inter-process communication. The strategy here was to keep part of the queue in the local PRQ and the other part of the queue in the PRQ object. A number of elements with the least scheduling time was kept in the local object. This minimized interface time because the local support PRQ returned the next element to be processed from its queue so long as its queue was not empty. Otherwise, the next element was obtained from the remote support PRQ.

Different queue sizes needed to be tested to determine how many elements should be kept in the local object. The interface time could be minimized, but how much depended on the queuing models that dictated the elements to be scheduled. Even though the local object kept a part of the queue, it could not schedule the elements independently from the PRQ; otherwise, the queuing discipline was not maintained. In some special cases, the local object needed to get the next element from the waiting process to determine in which queue the new element would be scheduled. By running the queuing models, it could be determined if this design was worthwhile.

4.3. Results for the Queuing Models

The implementation of the simulation of these queuing models consisted in the creation of the supporting simulation objects to test the environment partitioning approach on the Multimax. Many experiments were performed to test different object assignments to processors, and to determine some of the factors that could make the queuing models run faster. One of the primary goals of this research was to determine how many calls to the STAT object must be saved in the local STAT to minimize inter-processor communication time. To determine the batch size, the elapsed times for each simulation block were compared with the elapsed times obtained when running the simulation where no environment partitioning (regular program) was used. After several tests, where the simulation program created the local STAT object instead of the STAT object, it was found that the optimal number of calls to be saved was 50. If fewer than or more than 50 calls were transferred at a time, either too much time was spent in too many transfers, or too many bytes were transferred at a time, slowing down the transferring process. Another primary test was required to determine how large the queue would be in the local PRQ object. The queue size was changed for each run to determine the optimal size of 20. By keeping 20 entities in the local PRQ's queue, elapsed times as small as the regular program (no environment partitioning was employed) were obtained. No significantly advantage was obtained by creating the supporting objects for the PRQ.

4.3.1. Results for the MMc Queuing Simulation

The simulation queuing model program created two random number generator objects, one priority queue to be the event set, and one statistics accumulator object.

The first part of the study consisted of running the simulation program on one processor. No environment partitioning was employed. One of the test cases used 5 servers, an inter-arrival time of 10 (all times used are negatively exponentially distributed), and a service time of 35. Twenty blocks, each of length 1000 were run.

The elapsed time to execute a simulation block was 0.06 seconds. This time was the reference value to determine any speedup when environment partitioning was employed.

The second part of the study included creating support objects to determine the differences in the elapsed times. Each object was assigned to a different processor. There was a maximum of five processors needed and there were seven available. Therefore, the object assignment experiments were straightforward.

Figure 7 illustrates the speedups and processor efficiencies obtained from a selection of the experiments performed. Many other experiments were run, but the results shown are representative. The numbers presented are derived from the average execution time to run one block of the simulation.

The maximum speedup and the highest processor efficiency was obtained in Experiment B. Better elapsed times were obtained by creating the supporting STAT object. However, the local RNG and PRQ did not make any difference in the execution times.

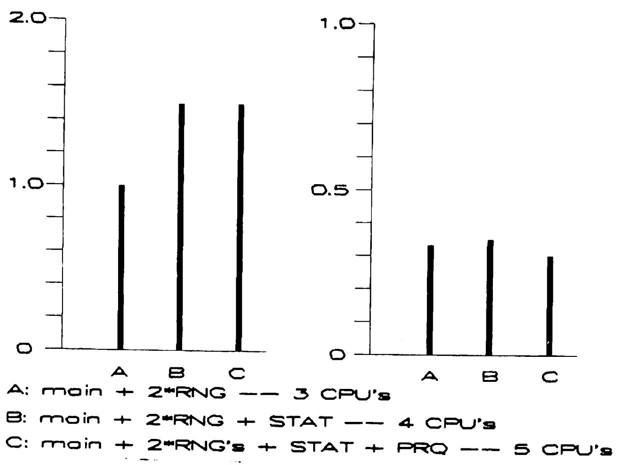

Figure 7. Speedups and Processor Efficiencies for Selected MMc Experiments

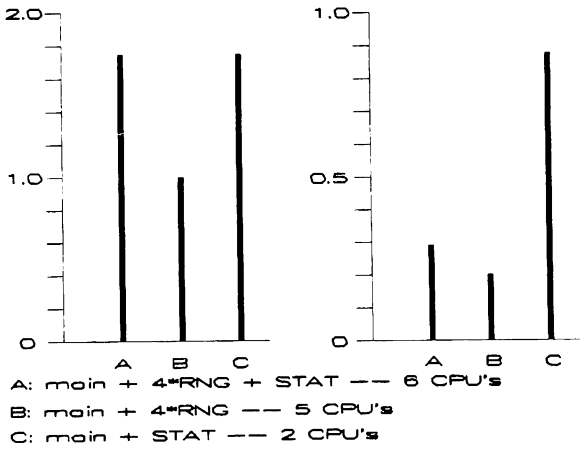

Figure 8. Speedups and Processor Efficiencies for Selected CR5 Experiments

4.3.2. Results for the CR5 Queuing Simulation

The implementation of the simulation of this queuing model consisted in the creation of the trivial supporting objects STAT and RNG. The trivial PRQ object was not created since the event size was not large. Environment partitioning was therefore not worth implementing for the PRQ object, because the queue was small. In the local STAT object, fifty calls were saved to minimize inter-process communication. Many experiments were performed to test different object assignments and determine the factors that made the simulation run faster.

The first part of the study involved running the simulation program on one processor with no environment partitioning. One of the test cases used 10 clients, a service time of 10 for the central (CPU) server, and service times of 30 for the other (I/O) servers. Twenty blocks, each of length 1000 were run.

The elapsed time to execute a simulation block was 0.06 seconds. This time was the reference value to determine any speedup when environment partitioning was used.

The second part of the study included the creation of four remote random number generators and one remote statistics accumulator. The other objects such as PRQ did not exploit environment partitioning. Each remote object was assigned to a different processor. There was a maximum of six processors needed in the simulation. Since there were seven CPUs available, the object assignments to processors was simple.

Figure 8 illustrates the speedups and processor efficiencies obtained from a selection of the experiments performed. Many other experiments were run, but the results shown are representative. The times are the average execution time to run one block of the simulation.

By just creating the supporting STAT object, better elapsed times were obtained. No advantage was shown when the supporting RNG object was created.

Experiment C obtained the best results for a block length of 1000. A substantial speedup was obtained by just creating the trivial STAT object as shown in Figure 8. There was no advantage in creating the trivial RNG object. The same behavior was obtained when running this queuing model with other input cases. Environment partitioning was applicable to one of the supporting objects when running the CR5 queuing model.

4.3.3. Results for the CQn Queuing Simulation

Environment partitioning was applied to this queuing model by creating n number of supporting RNGs and one STAT supporting object. Fifty calls were saved in the local STAT object to minimize the time spent in inter-process communication between the local STAT and the remote support STAT. Many experiments were executed to test different object assignments and determine any speedup.

The first part of the study involved running the simulation program on one processor without environment partitioning. The elapsed time to execute each block of the simulation was used to determine any speedup with environment partitioning. The elapsed time to execute a simulation block was 0.04 seconds.

The second part of the study included the creation of the remote STAT and RNGs. The number of remote RNG objects created depended on the number of servers. Each remote object was assigned to a different processor when up to seven objects were created.

One of the test cases used 3 servers, 40 clients, an service time of 30 for all servers. Twenty blocks, each of length 1000, were run.

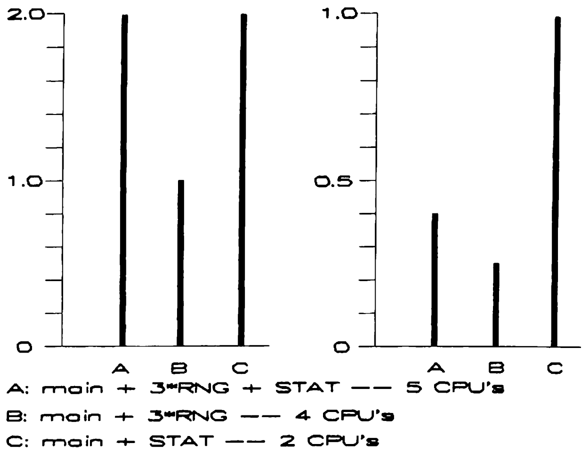

Figure 9. Speedups and Processor Efficiencies for Selected CQn Experiments

The largest speedup was obtained when creating the remote objects RNGs and STAT. In contrast with the other queuing models, the best speedup was not obtained by just creating the local STAT object. Some advantage was shown by creating the local RNG objects.

5. CONCLUSION

Speedups for the queuing systems MMc, CR5, and CQn were obtained when environment partitioning was applied. However, the speedups were not so significant as the ones reported by Comfort [Comfort 1991]. He estimated the performance of a distributed computed system performing the simulation of the queuing models instead of the implementation. The computer system he used consisted of n plus 1 8086 processors running at 9.54 MHz., and 640K of memory. He obtained a speedup of 6 for the MMc, 7.33 for the CR5, and 10.9 for the CQn. The speedups obtained when the queuing models were implemented on the Multimax were 1.5 for the MMc, 1.76 for the CR5, and 2.0 for the CQn.

Environment Partitioned Simulation provided a means of reducing the run time of queuing simulation programs. Speedups were obtained in the implementation of the simulation of the queuing models. However, the simulation supporting objects had knowledge of other objects' information and objective to minimize inter-processor communication. Because the interface was slow and the processors were very fast, the design of the objects to implement the simulation could not employ object oriented principles such as information hiding. No speedups would have been reported when implementing the queuing models simulation if all the object oriented principles were employed. The Multimax was not a suitable machine to exploit environment partitioning simulation.

Future studies include running the implementation of the queuing models simulation in other parallel machines and simulating the queuing models by incrementing the number of processors to determine the system behavior under different circumstances.

REFERENCES

Comfort, J. C., 'Environment Partitioned Distributed Simulation of Queuing Systems' ; **Proceedings of the 1991 Winter Simulation Conference**, Dec. 1991, pp. 584-92

AUTHOR BIOGRAPHIES

ISIS D. ROSELLO received her M.S. in Computer Science from Florida International University in 1993. She is presently a Senior Software Engineer at Coulter Corporation, in Miami.

JOHN CRAIG COMFORT is a Professor of Computer Science at Florida International University. His major research interest is parallel and distributed simulation. He has served in many capacities in the WSC, most recently as Program Chair, and in the Annual Simulation Symposium, most recently as President.