Proceedings of the 1994 Winter Simulation Conference
ed. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila

AN INTRODUCTION TO FAULT TOLERANT PARALLEL SIMULATION WITH EcliPSe

Felipe Knop
Edward Mascarenhas
Vernon Rego

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907, U.S.A.

V. S. Sunderam

Department of Math and Computer Science
Emory University
Atlanta, Georgia 30322, U.S.A.

ABSTRACT

This paper presents an overview of the *ACES* parallel software system and, in particular, an introduction to the *EcliPSe* layer of the system. The *ACES* system is a fault-tolerant, layered software system for heterogeneous-network based cluster computing. The *EcliPSe* toolkit, which resides on an upper layer, was constructed specifically for replication-based and domain-decomposition based simulation applications. It is not, however, restricted to simulations and supports any message-passing form of parallel processing. By taking advantage of networks of heterogeneous machines, generally "idle" workstations, *EcliPSe* programs can achieve supercomputer level performance with little programming effort – that is, low programming effort was a motivating factor in *EcliPSe*'s design. We present an overview of key application-level features in *EcliPSe*, support for fault-tolerant simulation, and performance results for three simple but large scale and representative experiments.

1 INTRODUCTION

The *EcliPSe* software system was originally designed to support straightforward and semi-automatic concurrent execution of stochastic simulation applications in a variety of parallel and distributed environments (Sunderam and Rego 1991). Since its inception, *EcliPSe* has been successful in demonstrating the practical viability of executing replication-based or domain-decomposition based simulations on heterogeneous networks of processors. Indeed, an early prototype demonstrated prize-winning performance in the investigation of universal constants in a polymer physics application which executed on a country-wide network of 192 processors (Nakanishi, Rego, and Sunderam 1992).

Simulation is known to be computationally intensive, with typical applications often executing for hours or days on fast scalar supercomputers. To reduce execution times, researchers have suggested various techniques for multiprocessor-based simulation. Considerable attention has been given to *distributing* a model over a number of processors in order to speed up the generation of a single sample path, in particular for discrete-event simulation. Examples of this approach include the conservative (Misra 1986) and the optimistic (Fujimoto 1990) protocols of distributed simulation. In addition to the complexities of application-level and system-level software development for distributed simulation, performance is often adversely affected by synchronization overheads intrinsic to distributed systems.

An alternate but complementary approach to model distribution is model replication, which is the approach adopted by the *EcliPSe* toolkit. This fact was already recognized by simulation researchers investigating the statistical consequences of parallel sampling (e.g., see (Biles et al 1985) and (Heidelberger 1988)). Instead of distributing a single model over a number (say n) of processors, n replications of the same model are made to run on the n distinct processors. This is useful for the most general stochastic simulation paradigm: *several sample paths are required in order that a statement with some statistical basis can be made.* Observe that one cannot avoid replication even for executions based on model distribution.

Thus, model replication may also be used to complement model distribution in that successfully distributed models can be replicated for even better performance. This statement is particularly relevant for large n, because distributed simulation *cannot* guarantee performance improvements with increasing numbers of processors. It has been our experience that model replication often exhibits potential for better performance than model distribution simply because replications exhibit few or no data dependencies and do not force synchronization constraints.

The *ACES* project is an effort geared towards producing a software base atop *cluster computing systems* (Turcotte 1993), which are systems consisting of heterogeneous networks of workstations and hardware (possibly massively parallel) multiprocessors. The design of the *ACES* system was motivated by a need for easy experimentation and rapid computation on flexibly configured environments, and its continuing development is guided by the following goals:

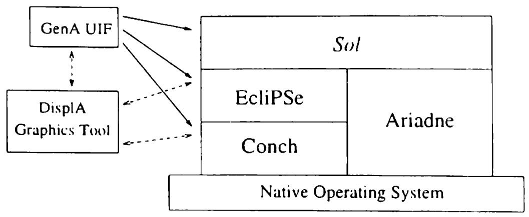


Figure 1: The *ACES* Software Architecture

ease of use: complex applications should be implementable in a high level manner.

portability: a distributed application should execute on a variety of architectures, including multiprocessors and heterogeneous workstations on wide area networks.

flexibility: the system should cater to a variety of applications, including replications, general data-parallel computations with interprocess communication and general distributed simulations or computations.

scalability: mechanisms that inhibit serializing bottlenecks should be provided, so that applications can scale well to run on a large number of processors.

fault tolerance: applications should be able to recover from machine crashes and other failures which are typical causes of unwanted termination for long-running executions.

The *ACES* system is currently organized in layers, as shown in Figure 1. The lowermost layer hosts a streamlined software base called *Conch*, providing higher layers with a virtual multiprocessor machine serviced by an efficient interprocess communications library. Given a set of heterogeneous machines and some user-specified topology, *Conch* builds a powerful multiprocessor environment where processes communicate with the aid of simple message-passing primitives.

Ariadne is an efficient lightweight process library designed to provide support for a variety of concurrency constructs at the *Conch*, *EcliPSe*, and *Sol* layers.

EcliPSe adds to the power of *Conch* by facilitating the design and implementation of replicative and domain-decomposed applications, and more importantly, by improving their performance.

The *Sol* (Simulation Object Library) system, which is resident at the uppermost layer, is a C++ based library that facilitates the construction of simulation models and other parallel applications in a variety of domains (Chung, Sang, and Rego 1993).

The remainder of the paper is focused on the *EcliPSe* layer, which is responsible for providing the base for efficient fault-tolerant simulations. The present version of *EcliPSe* is a robust and re-engineered version of prototype presented in (Rego and Sunderam 1992) and (Sunderam and Rego 1991), and has already been used in production applications, such as the work described in (Rintoul, Moon, and Nakanishi 1994). Section 2 presents an overview of *EcliPSe*, highlighting the main features. Section 3 presents a brief overview of its fault-tolerance features, and Section 4 describes a performance monitoring tool for *EcliPSe* applications. We present some experimental results in Section 5 and conclude briefly in Section 6.

2 OVERVIEW OF *EcliPSe*

2.1 Structure

A sequential application requires only minimal changes in order to utilize the power of *EcliPSe*. This generally entails insertion of *EcliPSe* primitives in the original source code with some (usually trivial) rearrangement of the code. Also, the user is required to provide a file containing the names of the machines to be used, usually (though not necessarily) "idle" workstations. The end result is a run consisting of a set of concurrently executing *sampler* processes coordinated by one or more *monitor* processes.

An *EcliPSe* program must contain the following components:

Computation code. This is code that is run by each of the *samplers*, being responsible for most of the "actual work" that the application performs. The computation code usually requires (input) data from and returns (result) data to a monitor process.

Monitor function(s). These are functions executed by *monitor processes*. A monitor is responsible for coordinating the computation done by a set of samplers, generating data for and collecting data from these processes, and finally terminating the computation.

Declarations. Each type of data item that is exchanged between monitors and samplers must be declared. By declaring data types, the user hands over to *EcliPSe* the task of data handling. Declarations provide the added advantage of making explicit the flow of information between monitors and samplers.

2.1.1 Declarations

EcliPSe declarations are handled by a special preprocessor, allowing users to make declarations using a "C-like" syntax. For example, suppose that samplers produce an array of 10 double precision numbers for the monitor. The declaration of this data item is simply:

```
eclipse_decls {
        double   type_result[10];
}
```

The `eclipse_decls` block defines the region that the preprocessor acts on. The preprocessor declares an *integer* variable called `type_result` that, at run time, will contain a handle used in all subsequent *EcliPSe* calls that refer to the double precision array (analogous to the notion of "file descriptor" in UNIX systems). Therefore, when an array of 10 double precision numbers is to be sent to the monitor, only the data type handle and a pointer to the data need be provided. Data is then transmitted in a machine-independent format, which is crucial for computation on heterogeneous machines.

2.1.2 Computation Code Primitives

The basic primitives available to samplers are simple: `request_data` obtains data from a monitor, and `put_stat` sends data to a monitor. Both take as parameters an (integer) type handle obtained in the declarations and a pointer to the data to be transferred. In general, if an application's sequential code is already available, changing it to work with *EcliPSe* is a relatively simple task. It suffices to (a) replace the data input code (sometimes obtained from the keyboard or from a file) by the corresponding `request_data` primitives, and (b) replace the collection of result and statistics by the corresponding `put_stat` primitives.

2.1.3 Monitor Function

All code related to file I/O and to the collection of statistics is placed inside the monitor function, which is executed by a *monitor process*. If a sequential application is being ported to *EcliPSe*, writing the monitor generally means moving the above functionality from the computation code into the monitor. Calls to `produce_data` (counterpart to `request_data`) and `collect_stat` (counterpart to `put_stat`) are inserted when needed.

Table 1 summarizes the main primitives available to monitors and samplers.

2.2 General Features

Without mechanisms for efficient data transfers between samplers and monitors, it is possible for sampler-generated data to cause a bottleneck at a monitor. *EcliPSe* provides a set of control mechanisms that prevent such serializing bottlenecks and network clogs from occurring. These include:

Granularity control. With a small change in a data type declaration, the user may specify a "grain size" to be used for that type. As a result, subsequent `put_stat` calls buffer data instead of sending data directly to a monitor. When the number of buffered data items reaches "grain size", the buffered data is sent in a single message. This helps reduce network usage and also decreases overhead at monitors and at samplers.

Multiple monitors. If a monitor is being overworked due to high incoming traffic, then the incoming workload can be distributed among several monitor processes. This is accomplished by coding additional monitor functions for the different workloads, specifying their names in the declarations, and indicating (again, in the declarations) which data types are to be associated with which monitors. Code for the sampler need not change.

Tree-combining. If a monitor were to receive data directly from a large number of samplers, the amount of incoming traffic and resulting "combining work" could make the monitor a bottleneck for the entire computation. This can happen, for example, when the monitor averages results it receives from samplers. To prevent such a bottleneck from occurring, *EcliPSe* allows processes to be organized in a virtual tree structure, with a user-defined topology and the monitor as the root. Each sampler transparently sends data to its parent in the tree, instead of sending data directly to the root. Each such parent *combines* its own results with the results it receives from all of its children in the tree, applying the same operation that the monitor process would have applied had the tree-combining scheme not been used. As a result, the monitor at the root only needs to combine data it receives from its own children.

The user may choose to employ the tree-combining scheme by declaring a data type to be a "combining type" and by specifying its corresponding combining operation. The latter may be either a user-written function or one of the standard combining operations provided by *EcliPSe* (i.e., averaging, summation, concatenation, and others). No change is required in the code for samplers.

Data-diffusing. The virtual tree structure described above can also be used to speed up the distribution of data from a monitor to each sampler. Instead of sending data directly to each sampler, a monitor only needs to use the `produce_data_diffuse` primitive on an array of data items to be distributed. Data is then "diffused" down the tree, with each sampler receiving a data item. The process behaves like the tree-combining scheme in reverse. The same primitive can also be used for an efficient data broadcast.

3 FAULT TOLERANCE

Typical *EcliPSe* applications often require large computational resources, which is sometimes translated to mean tens or hundreds of machines executing continuously for several days, and possibly weeks. In this setting, the need for fault tolerance is critical, because heterogeneous distributed computing in an open, uncontrolled environment is generally unreliable. Without fault tolerance, long-running applications may never run to completion.

We have attempted to address most of the typical problems that occur during execution of a large-scale application and incorporated our solutions in the *EcliPSe* toolkit. Recovery is attempted whenever it is meaningful. The following is a list of problems detected, with corresponding actions.

1. **Process or machine failure**. In general these are caused by operating system resource exhaustion

Table 1: Main Primitives Available in *EcliPSe*

PRIMITIVE	MEANING
COMPUTATION CODE	
request_data(int type_id, char *ptr_data)	Receive one data item from the monitor
put_stat(int type_id, char *ptr_data)	Produce one data item for the monitor
MONITOR FUNCTION	
produce_data(int pr_id, int type_id, char *ptr_data)	Produce one data item for a sampler
produce_data_diffuse(int type_id, char *ptr_data)	Produce data items for all samplers using the diffusion scheme
collect_stat(int pr_id, int type_id, char *ptr_data)	Collect one data item from a sampler; pr_id may be ANY_PROC
collect_stat_combine(int type_id, char *ptr_data)	Collect the combined data items from all samplers
MISCELLANEOUS	
get_first_proc(), get_next_proc(), is_end_proc()	Process listing primitives
get_first_child(), get_next_child(), is_end_child()	List children
get_parent()	Return id of parent process

(caused by an *EcliPSe* application or an alien application using one of the *EcliPSe* machines), operating system error, hardware fault, machine shutdown, or network failure. The default action in each case is to run a replacement process that substitutes for the failed process.

2. **Software exceptions**. These are exceptions detected by the hardware/operating system and are usually due to application programming errors. Executing a replacement process is not wise since the error is likely to repeat itself unless corrected by the application programmer. Only an appropriate error message is printed and the application is typically terminated.
3. **Infinite loops in the application**. Even if the application successfully passes small-scale test runs, problems may arise with large-scale runs. A problem that might occur is that one or more samplers enters an infinite loop because of a subtle programming error at the application level. Alternately, a sampler may appear to be in an infinite loop if it works slowly, due to a hitherto undetected load surge on its host machine or because it has been given too low a priority by its host's scheduler. A hard problem to tackle in the general case, detection of infinite loops is performed using some user-guided heuristics. Upon detection of such a situation, appropriate user action is taken, with the default being simply a warning message.

To allow recovery from a process or machine failure, a checkpoint-rollback mechanism is used: data from all processes is periodically saved, and then later restored should a failure occur, with a new process being created to replace that which failed. While *EcliPSe* checkpointing is not transparent to the application, it requires little programming effort and has the added advantage of being low cost and efficient; the user only needs to declare what must be saved and specify the few points in the program where checkpoints should occur. To specify the data to be saved, the user must declare a set of *recovery data types* together with a set of data pointers. As an example, a 20-element integer state vector is saved by all samplers at a checkpoint, and then restored in a rollback, with the following declaration:

```
int type_ft_proc[20] (ft_proc <state_array>);
```

where `state_array` is a user-provided pointer to the data being saved. A similar array used for saving and restoring a monitor's state is declared by replacing `ft_proc` by `ft_mon` in the type declaration above.

The user indicates the points in the program where checkpoint and recovery should occur by inserting calls to `check_recover()`. After this is done, checkpoints and rollbacks occur without further user intervention.

Provided adequate state vectors are specified, rollbacks still allow the program to produce the same output regardless of the number of process failures even when samplers base their computations on random numbers.

A shutdown-restart mechanism has also been provided: the user may stop the application and restart it later, possibly using another set of machines (even of a different type).

By taking advantage of the application's structure, *EcliPSe* programs can often achieve an almost negligible checkpoint overhead, which in turn drastically reduces the fault tolerance performance penalty. The details of how this is accomplished are presented in (Knop, Rego, and Sunderam 1994b).

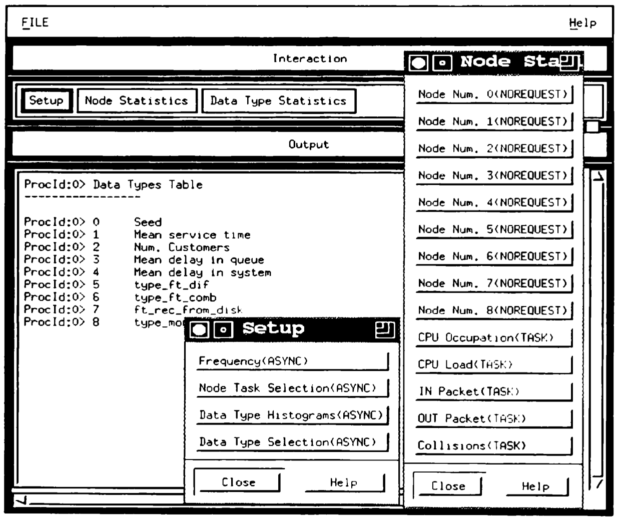
4 PERFORMANCE MEASUREMENT

EcliPSe applications support a variety of computation structures (see (Knop, Rego, and Sunderam 1994a)) and execute on a number of machine environments. Bottlenecks in a distributed application, however, may impair execution performance, resulting in a waste of computational resources. Early experience with some *EcliPSe* applications indicate that bottlenecks tend to occur when (a) the monitor is overworked by a high influx of data messages, and (b) samplers have to share a host CPU with other non-*EcliPSe* applications. Some bottlenecks are easily circumvented by a small change in the program or execution environment, provided the source of the bottleneck can be found.

To address the problem of locating execution bottlenecks, the *ACES* system provides a tool for the interactive

Table 2: Performance Statistics Collected by the Tool

STATISTIC	MEANING
NODE STATISTICS	
CPU occupation	Fraction of time this node uses the CPU
CPU load	Average number of processes running or waiting for the CPU
Input packet rate	Rate of incoming network packets at the node
Output packet rate	Rate of outgoing network packets at the node
Collision rate	Rate of packet collisions for bus-like networks
DATA TYPE STATISTICS	
Messages received or sent	Accumulated number of data messages received or sent for this type
Messages pending	Messages of a specific data type queued in this process's input buffer
Waiting time	Time (cumulative) this process waits for data of a specific type



```
ProcId:0> Data Types Table
-----------------

ProcId:0> 0     Seed
ProcId:0> 1     Mean service time
ProcId:0> 2     Num. Customers
ProcId:0> 3     Mean delay in queue
ProcId:0> 4     Mean delay in system
ProcId:0> 5     type_ft_dif
ProcId:0> 6     type_ft_comb
ProcId:0> 7     ft_rec_from_disk
ProcId:0> 8     type_mo
```

Figure 2: Main Window of Display Tool

display of graphical performance data collected and displayed on-the-fly during an application run. Two types of statistics are periodically collected for each process: node statistics and data type statistics. The former refers to machines which host *EcliPSe* processes, and the latter refers to *EcliPSe* data types in which the user has particular interest. Table 2 shows a list of parameters that the tool currently displays. Statistics from all processes are displayed as histograms, making it easy for the user to spot bottlenecks. Use of this tool requires no changes in the application program.

The main window of the tool is shown in Figure 2. The buttons shown under the title *Interaction* allow a user to interact with an ongoing *EcliPSe* application. Using the *Setup* panel and clicking on options available, a user can select/deselect the plotting of histograms for CPU occupation-level, CPU load, in/out packet rate, etc. Figure 3 shows an example of a histogram plotted for the CPU occupation-level of an M/GI/1 queue application utilizing eight samplers and one monitor (node # 0).

More details about the performance tool can be found

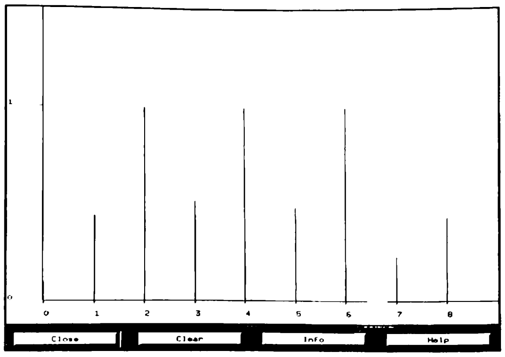


Figure 3: CPU Occupation-level Histogram for 8 Processors and 1 Monitor

in (Knop et al. 1994).

5 EXPERIMENTS

To give the user an idea of *EcliPSe*'s performance on easily understandable examples, we present the results of a few experiments. The goal is to demonstrate the utility of some of the main features described in the paper, and to give some indication of *EcliPSe*'s performance on typical applications with different computation structures.

5.1 Machines

The experiments were conducted on a network of 177 SUN IPC SPARC workstations. These machines reside on four distinct local area networks, all connected via a single gateway. The communication time (as measured by the UNIX `traceroute` command working with 40-byte packets) between local machines is 2 ms, and this increases to 3 ms when a message has to pass through the gateway. Each machine is rated at 15 MIPS.

5.2 Programs

5.2.1 M/G/1 Queue Simulation: Replication

In this program (hereafter referred to as **mg1**), each replication simulates an M/GI/1 queue for a fixed number of arrivals. Based on batch-means, the statistics sent to the monitor include mean system delay and maximum number of customers found in the queue. The regenerative method may also be used (as was done in (Rego and Sunderam 1992)) to estimate these quantities. The monitor utilizes results from independent parallel replications to build a confidence interval for both statistics. Though times between the reporting of samples can be large, the monitor may be overworked if a large number of processes is used. For the experiments reported in this paper, the total number of samples collected was fixed. Two variations of the program were tested: one using the tree-combining scheme and one not using this scheme.

5.2.2 Multidimensional Integral Estimation

This application (hereafter referred to as **integral**) is an example of the use of the *sample-mean* Monte Carlo method in estimating multidimensional integrals. In this particular experiment, we estimate

$$\int_0^1 \int_0^1 \cdots \int_0^1 e^{\sum_{i=0}^{d-1} x_i^2} dx_0 dx_1 \ldots dx_{d-1}$$

with $d = 20$, to demonstrate the effectiveness of Monte Carlo in high-dimensions.

In the **integral** procedure, each sampler repeatedly chooses a random point $(x_0, x_1, \ldots x_{d-1})$ inside the region over which the integral is defined and computes the value of the function at this point. The resulting value is sent to the monitor using a `put_stat` primitive, with the monitor averaging all results it receives. Since the computation time for a single sample is small, both a tree-combining scheme and "grainsize" larger than 1 were used to avoid overworking the monitor.

5.2.3 Absorption Times in Markov Chains

Given a $(k+1)$-state, discrete time Markov chain and its associated transition probability matrix P, program **absorb** estimates the average number of steps required to take the chain from state k to the absorbing state 0 (i.e., the time to absorption). The monitor builds matrix P and broadcasts it to all samplers. Upon receiving P, the samplers simulate independent realizations of the time to absorption and sends results to the monitor, with a tree-combining mechanism used to average results. The monitor is then able to compute an estimate for the average time to absorption.

The parameters used in this experiment were $k = 256$ and `grainsize` $= 128$. A total of $2^{18} = 262144$ samples are generated in all by the samplers. A fault-tolerant version of the application was used, based on the fault tolerance interface outlined in Section 3. For fault-tolerance, a state size of 8 bytes was used for each sampler, with 36 bytes for the monitor.

5.3 Results

5.3.1 Effect of the Tree-Combining Scheme

mgl was chosen for the evaluation of the tree-combining scheme. In the version without tree-combining, samples of average delay on the system and samples of maximum number of customers in queue are sent directly to the monitor, which utilizes a running-mean procedure to compute sample mean and variance. In the tree-combining version, the **average** combining operation is used, together with the *EcliPSe* monitor function which accumulates results. For both programs, 25000 customers were simulated in each replication, for a total of 2048 replications. Figure 4 shows the execution time for both versions of the program, while

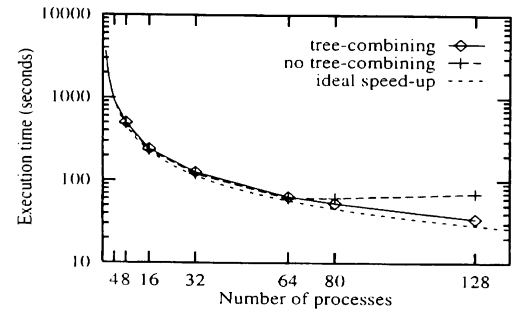


Figure 4: Execution times for M/GI/1 Application

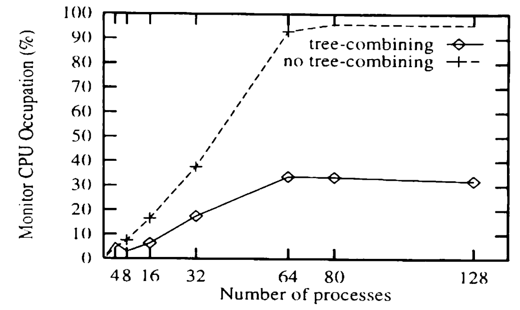


Figure 5: Monitor CPU Occupation Level for M/GI/1 Application

Figure 5 shows the corresponding CPU occupation-level of the monitor.

For fewer than 64 processes, the "normal" combining program was slightly (up to 5%) faster. This happens because of combining overhead imposed on the samplers. For a larger number of processes, the "normal" combining program was not able to take advantage of the extra samplers, and an explanation for this can be found in Figure 5: the monitor is overloaded with work generated by a high influx of messages.

This experiment demonstrates the usefulness of the tree-combining scheme when a large number of processes is used. It also demonstrates that use of tree-combining must be made with care, to avoid unnecessary overhead.

5.3.2 Effect of Different Tree Topologies

With tree-combining, use of different tree topologies can lead to different performance characteristics. To demonstrate this effect, we experiment with five different topologies. As an application we use **integral** on 64 samplers, each using grainsize of 1000, with a total of 8 million samples generated by the system.

To describe each tree topology, the following notation is used. A $(x_1, x_2, \ldots)$-tree is a tree where the root has x_1 children, each of which has x_2 children, and so on. Figure 6 shows the program execution times for the following topologies: $(16, 3)$, $(24, 1\ or\ 3)$ (in this topology each of the 24 nodes in the first level has either 1 or 3 children, so that the total number of nodes is still 64), $(8, 7)$, $(4, 15)$, and $(4, 3, 4)$. Figure 7 shows the corresponding

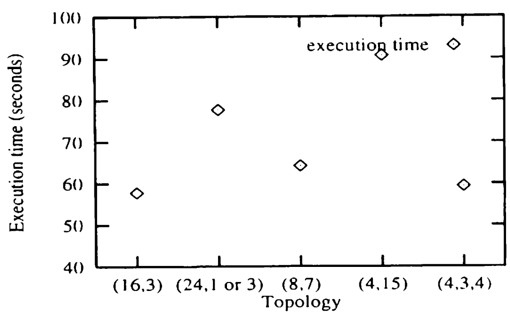


Figure 6: Execution Times for the Different Tree Topologies

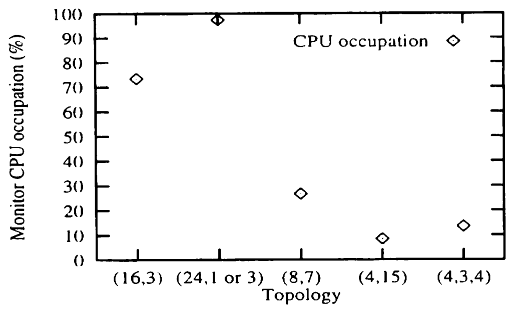


Figure 7: CPU Occupation-Level of Monitor for Different Tree Topologies

CPU occupation-level of the monitor.

The values of the monitor's CPU occupation-level are consistent: the load of the monitor tends to be proportional to the number of children it has. Using the limited data provided by Figure 6, we may arrive at the following conclusions: (a) tree topologies that overwork the monitor (like $(24, 1 \, or \, 3)$ for the present experiment) result in poor speedup, and (b) tree topologies that avoid overworking the monitor at the expense of assigning several child processes to each non-monitor process (like $(4, 15)$) also behave poorly.

5.3.3 Effect of the Grainsize

The program chosen for the "grainsize" experiment was **integral**, using the tree-combining scheme. The total number of samples was 8192000, and the program was run with 64 processes placed in a $(16, 3)$-tree. Figure 8 shows the program's execution times for grainsize varying from 100 to 51200, while Figure 9 shows the corresponding monitor occupation-level of the CPU.

For a grainsize smaller than 800 the program's execution time increases roughly proportionally with the inverse of the grainsize, which can be attributed to the monitor not being able to handle the incoming data. Data in Figure 9 indicates that with grainsize 800 the monitor is using almost all CPU time available, and that the total amount of work accomplished will only decrease if the grainsize is made less than about 800.

For a grainsize larger than 800 the monitor load is much

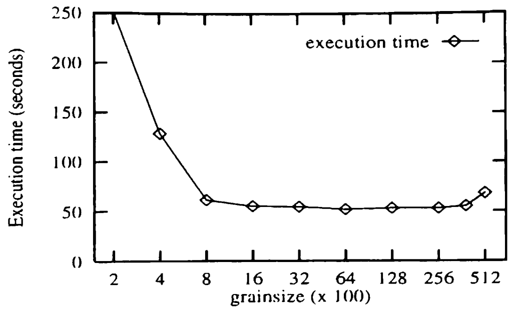


Figure 8: Execution Times for the Different Values of Grainsize

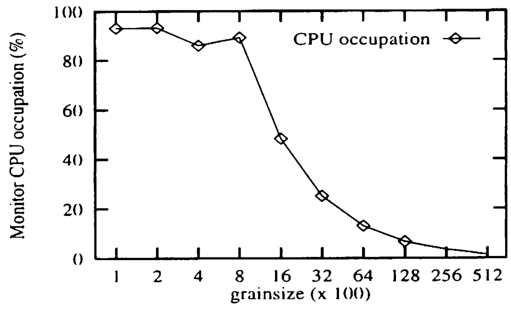


Figure 9: Monitor CPU Occupation for the Different Values of Grainsize

less of a factor, and ordinarily the program's execution time should decrease until an asymptote is reached. An explanation of the unexpected increase in execution time for a very large grainsize may be found in the program's pattern of memory usage: a larger grainsize requires a large amount of memory to be used by the samplers to store samples, therefore causing an increased number of page faults.

5.3.4 Fault Tolerance Overhead

Program **absorb** was run with different numbers of processes to evaluate the effects of checkpointing overhead. Execution times are shown in Figure 10.

From this figure it is clear that the application is still able to obtain good speed-up, despite frequent checkpoints and the significant proportion of time (about 20% of the total execution time for 128 processes) spent by the monitor

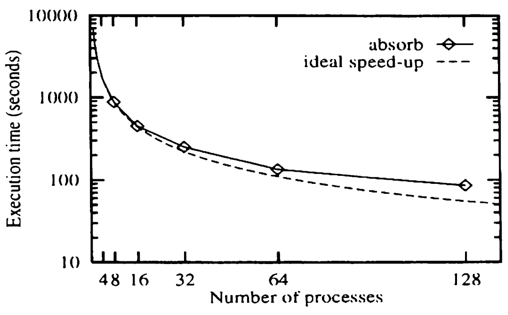


Figure 10: Execution Time for the **absorb** Program

to broadcast the transition probability matrix. This graph shows that fault-tolerant applications in *EcliPSe* can be developed with negligible checkpoint overhead, using low programming effort to make the program fault-tolerant.

6 CONCLUSIONS

Our experiments with the *EcliPSe* system have led to interesting results. Simple fault-tolerant applications can be designed and made to run on a network of dozens of machines in a matter of a few hours. Such programs are fairly easy to understand and modify, and perform very well in a multi-machine run. This indicates the effectiveness of *EcliPSe* in speeding up the execution of model replication applications.

Working with *EcliPSe* also gives us a feel for possible improvements. A graphical tool is being designed (Mascarenhas et al 1994) to allow the user to specify the monitor in a high level manner, with the entire monitor function produced automatically. Also, our experiences and empirical results with *EcliPSe* indicate that automating the virtual machine configuration procedure may result in improved application performance.

ACKNOWLEDGMENTS

This research was supported in part by NATO-CRG900108, NSF CCR-9102331, ONR-9310233, and ARO-93G0045. The first author was supported by CNPq-Brazil, process number 260059/91.9.

REFERENCES

Biles, B., D. Daniels, and T. O'Donnell. 1985. Statistical considerations in simulation on a network of microcomputers. In *Proceedings of the Winter Simulation Conference*, 388–393.

Chung, K., J. Sang, and V. Rego. 1993. *Sol-es*: An object-oriented platform for event-scheduled simulations. In *Proceedings of The Summer Simulation Conference.*

Fujimoto, R. M. 1990. Optimistic approaches to parallel discrete event simulation. *Transactions of the society for computer simulation*, 7(2):153–191.

Heidelberger, P. 1988. Discrete event simulations and parallel processing: statistical properties. *SIAM Journal on Scientific and Statistical Computing*, 9:1114–1132.

Knop, F., V. Rego, and V. Sunderam. 1994a. EcliPSe: A system for fault-tolerant replicative computations. In *Proceedings of the IEEE/USP International Symposium on High-Performance Computing.*

Knop, F., V. Rego, and V. Sunderam. 1994b. Failure-resilient computations in the EcliPSe system. To appear in *Proceedings of the International Conference on Parallel Processing.*

Knop, F., E. Mascarenhas, V. Rego, and V. Sunderam. 1994. Fail-safe concurrent simulation with EcliPSe: an introduction. Submitted for publication.

Mascarenhas, E., *et al.* 1994. GenA: A Gui for Generation of ACES Applications. Technical report, Purdue University (in preparation).

Misra, J. 1986. Distributed discrete-event simulation. *Computing surveys*, 18(1):39–65.

Nakanishi, H., V. Rego, and V. Sunderam. 1992. Superconcurrent simulation of polymer chains on heterogeneous networks. *1992 Gordon Bell Prize Paper, Proceedings of the Fifth High-Performance Computing and Communications Conference: Supercomputing '92.*

Rego, V. J., and V. S. Sunderam. 1992. Experiments in concurrent stochastic simulation: the *EcliPSe* paradigm. *Journal of Parallel and Distributed Computing*, 14(1):66–84.

Rintoul, M. D., J. Moon, and H. Nakanishi. 1994. Statistics of self-avoiding walks on randomly diluted lattice. (to appear) *Phys. Rev. E.*

Sunderam, V. S., and V. J. Rego. 1991. *EcliPSe*: A system for high performance concurrent simulation. *Software-Practice and Experience*, 21(11):1189–1219.

Turcotte, L. H. 1993. A survey of software environments for exploiting networked computing resources. Technical report, Engineering Research Center for Computational Field Simulation, Mississippi State University.

AUTHOR BIOGRAPHIES

FELIPE KNOP is a Ph.D. student in Computer Sciences at Purdue University. He received a Masters degree in Computer Sciences from Purdue University in 1993 and a Masters degree in Electrical Engineering from University of São Paulo, Brazil, in 1990. His current research interests include parallel and distributed simulation, and multiprocessor operating systems.

EDWARD MASCARENHAS is a Ph.D. student in Computer Sciences at Purdue University. He received a Masters degree in Computer Sciences from Purdue University in 1993 and a Masters degree in Industrial Engineering from NITIE, India. Current research interests include parallel computation, distributed simulation, user interfaces and multi-threaded programming environments.

VERNON REGO, Ph.D., Michigan State University, East Lansing, 1985, is an Associate Professor of Computer Sciences at Purdue University. His research interests include parallel and distributed computing environments, networks, computational probability and stochastic simulation. He was awarded the 1992 Gordon Bell Prize for parallel processing.

V. S. SUNDERAM, Ph.D., University of Kent, 1986, is an Associate Professor in the Dept. of Math. and CS at Emory University, Atlanta. His research interests include parallel processing paradigms, specifications and tools, heterogeneous distributed systems, communication protocols and simulation. He was awarded the 1992 Gordon Bell Prize for parallel processing.