1991

# Experiments in Concurrent Stochastic Simulation: The EcliPSe Paradigm

Vernon J. Rego
*Purdue University*, rego@cs.purdue.edu

V. S. Sunderam

Report Number:
91-005

# EXPERIMENTS IN CONCURRENT STOCHASTIC SIMULATION: THE ECLIPSE PARADIGM

Vernon J. Rego
V. S. Sunderam

CSD-TR-91-005
January 1991
(Revised May 1991)

# Experiments In Concurrent Stochastic Simulation: The *EcliPSe* Paradigm*

Vernon J. Rego†
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

V.S. Sunderam
Department of Math and Computer Science
Emory University
Atlanta, GA 30322

**Abstract**

This paper presents results on the performance of a novel and flexible concurrent simulation environment known as *EcliPSe*. The paradigm we advocate is based on the premise that replication based simulations, either non-distributed or minimally distributed, yield excellent speedups. The approach used makes concurrent simulation easily accessible to researchers because its use does not require knowledge of parallel programming. The experiments we report include Monte-Carlo type simulations (e.g., estimating integrals, order-statistics), Markov-chain simulations (hitting-times, distributed algorithms), and discrete-event simulation (e.g., tail probabilities in queues, FDDI token ring performance, and simulations of high performance software testing techniques on SIMD machines).

*This research was performed at the Mathematical Sciences Section of Oak Ridge National Laboratory under the auspices of the Faculty Research Participation Program of Oak Ridge Associated Universities, and supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. DOE, under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

†Also supported in part by the National Science Foundation, under Grant No. ASC-9002225

# 1 Introduction

Since its use at Los Alamos during World War II, stochastic simulation [43] has gained increasing acceptance as a viable tool for understanding the behavior of complex phenomena. In a recent article on High Performance Computing [2], Gordon Bell describes "Large-Scale Scientific Experiments Based on Simulation," to be one of six key application areas. Nobel Laureate Ken Wilson characterizes [2] computer simulation as the third paradigm of science, supplementing theory and experimentation, a paradigm shift that will transform every facet of science, engineering and mathematics.

Stochastic simulation is presently used in a variety of applications such as

- molecular dynamics methods in chemical physics [49]
- computationally intense lattice theory problems [15, 16] in statistical physics,
- computation of Feynman path-integrals [12, 49],
- Monte Carlo methods in the study of liquids [3, 25],
- placement of VLSI circuit components [18],
- global optimization and random search [44],
- Markov random field texture models [9],
- stochastic models of computer and communication systems [11, 29],
- battle management models [33],
- belief networks and influence diagrams in AI [36],
- genetic algorithms [19] and simulated annealing [1, 26], and
- Monte Carlo solutions to matrix problems [44].

to name just a dozen select applications.

In essence, stochastic simulation is a technique that allows an analyst to mimic a stochastic process of interest. Using pseudo random numbers, a high speed computing machine can be made to yield realizations of a random process several times faster than working directly with a physical process. In many instances speed is not the only factor since the cost of working with a physical process may be prohibitive. Stochastic simulation is therefore an invaluable aid in studying stochastic phenomena, a tool that is likely to grow in importance

with the complexity and size of problems which, not yielding to deterministic methods, must be solved via sampling.

Even though the speeds of modern uniprocessors now make possible simulations that were either too time-consuming or simply impossible to conduct a decade ago, there is an ever present need for reducing model execution time. Over the past decade, a significant amount of attention has been devoted to *distributing* a model over a number of processors in order to speed up the generation of sample paths, in particular for discrete-event simulations [17]. This paper attempts to address a broader class of simulation models, loosely called *concurrentizable* stochastic simulation models, which can benefit substantially from the power of multiprocessing.

We define a stochastic simulation model to be *concurrentizable* if either independent or dependent replications of the model can be executed concurrently. As a simple example in the independent case, consider a simulated annealing model [1] operating on a domain $D$. If the domain $D$ can be partitioned into disjoint subdomains, say $D_1$, $D_2, \ldots, D_N$, where $D = D_1 \cup D_2 \cup \ldots \cup D_N$, replications of the same model can be made to execute independently on independent processors or sets of processors with the intention of determining that specific region $D_k$ containing the global optimum. Observing that each subdomain is being operated on independently, this kind of concurrency is otherwise termed data parallelism [15]. Dependence between processors can be induced in the following manner. In looking for the global optimum in $D$, processors can be required to exchange information about their domains, converge on the "best" subdomain $D_j$, subdivide it for data parallelism, and then repeat the annealing algorithm on their respective subdivisions of domain $D_j$. This algorithm would terminate either when a certain cost has been exceeded or when the optimum has been found. In both the independent as well as the dependent case, the replicated simulations execute concurrently. This concurrentizing strategy works equally well for a variety of stochastic simulations, as will be shown in Section 5.

In our notion of concurrent simulation, no restriction is placed on the number of processors required to execute a single model. The emphasis is on model *replication* to achieve natural parallelism, and model *distribution* is of secondary importance. Whenever possible, the entire model is made to execute on a single processor; and whenever a model is required to be distributed across processors, the number of processors involved in executing the model is kept to a minimum. Various theoretical and practical problems arise in such replication based concurrency (e.g., see [21, 46]). For example, care must be taken in ensuring that the estimators constructed by the different sampling processors yield unbiased estimates; obvious uniprocessor estimators are known to possess poor convergence properties [21] in multiprocessor settings. Perhaps such pitfalls are reasons for a conspicuous absence of rigorous empirical research in, or the development of a tool for, concurrent simulation systems.

In this paper, we report the performance of a high performance replication based con-

current simulation system, known as *EcliPSe*, on a suite of eight simulation experiments drawn from various application domains. *EcliPSe* was designed and implemented mainly for stochastic simulation applications, and we emphasized ease of software system use in its construction. *EcliPSe* takes as input a sequential simulation model, along with user-defined constructs such as routines for pooling statistics, and replicates the sequential model across processors; the toolkit's flexibility lies in its ability to cater to a variety of simulation problems (e.g., Monte Carlo, discrete-event simulation, stochastic optimization), a variety of multiprocessing schemes (e.g., replication, distribution, hybrids), and a variety of supporting architectures (e.g., shared memory, distributed memory, heterogeneous wide area networks). Though still in its infancy, we believe that the *EcliPSe* system [46] is a promising step in the direction of concurrent simulation software.

The remainder of this paper is organized as follows. In Section 2 is presented a brief commentary on related work in the area of multiprocessor simulation. In Section 3 is presented our motivation for designing and implementing the *EcliPSe* system the way we have, and Section 4 contains some of the salient design aspects of the system - summarizing details given in [46]. In Section 5 we outline several simulation experiments that were run using *EcliPSe*. In each case, programming ease, portability and near-linear speedup gave ample proof of the high-performance characteristics of the toolkit; some of this is apparent from the timing results given in the same section. Section 6 contains a brief conclusion.

## 2 Related research

In recent years, researchers have suggested various ways of using shared and distributed memory MIMD machines for performing simulation. The computer scientists' approach has been to focus on discrete-event simulation problems requiring model distribution [17], where a model is functionally decomposed and executed across several processors. As a result, the model replication aspect appears to have been largely neglected. Apart from more theoretical studies [4, 21] on model replication, there has been little experimental work measuring performance and virtually no empirical work on software systems reported in the literature for replication based simulation models. Though it has been suggested [17] that replication is practical only for largely stochastic models, there is no empirical evidence to support this view. In addition this view can be misleading because a metric of model stochasticity is generally not a concern, and a variety of models including discrete-event models may be set in a purely stochastic framework.

There are many unresolved issues in realizing effective distributed simulation software systems, even though there has been prolific research in the more theoretical aspects of such simulation systems. While this phenomenon is not surprising, especially considering the rich nature of problems relating to simulation on distributed systems, there is a noticeable lack

of software available to the user community for performing distributed simulation.

An important reason for viewing model distribution very favorably is that there is no alternative when the model to be run exceeds the capacity of a single host processor. Nevertheless, present-day processors are individually capable of hosting enormous problems and, for such problems, are likely to make replication based simulation a viable strategy. In addition, even if a model has to be distributed across a set of processors, one can still replicate the distributed model across several sets of processors, so that the two strategies may complement one another. The last point becomes especially important when one observes that that there is a limit to how much a given model can be distributed without an adverse effect on system performance. Consider for example a token ring model with $N$ stations. A logical approach to distributing the model would be to place each station on a processor, or perhaps two processors if station functions warranted additional processing power. When $N = 50$, distributing the model across 1024 available processors of a hypercube would leave between 90% and 95% of the multiprocessing power unused.

Another important reason for pursuing model distribution is positive speedup, but it is shown here that our concurrent stochastic simulation approach yields excellent speedups with added advantages such as ease of model implementation and ease of portability across machine architectures. Fujimoto [17] gives a good survey of techniques and problems associated with distributed simulation. It is interesting to note that the state-of-the art in distributed simulation invariably involves discrete-event simulation problems, in particular queueing network problems, and MIMD architectures with either shared memory or distributed memory, but not both architectures treated together.

Queueing network abstractions of computer systems and computer network problems have always been a focal point in the performance modelling community, and it is perhaps this reason even more than its discrete-event nature that has brought queueing into the distributed simulation application domain. Queueing network problems are excellent candidates for testing out ideas in model distribution. Unfortunately, there are many simulation problems that do not lend themselves to queueing abstractions, and consequently the excessive emphasis on distributed queueing network simulation is perhaps misdirected. Though not queueing network abstractions, several of the application areas mentioned in Section 1 can be viewed as discrete-event problems but are largely ignored by research in distributed simulation.

Much of the current work in distributed simulation favors MIMD architectures, and apparently more so the distributed memory architectures than the shared memory architectures. There have been some negative reports on the performance of shared memory machines for distributed simulation [38, 39]. We speculate that a possible reason for less attention being given to them is the limited number of processors that are available on such machines (e.g., Sequent). On the other hand, the distributed memory hypercubes or networks of workstations have a much larger processing and expansion capability (e.g., Intel

iPSC/860).

There has been some recent work which attempts to correct some of the negative views of using shared memory machines for distributed simulation [48]. Proposed distributed simulation systems (e.g., [8], or Synapse [48]) usually choose either shared or distributed memory architectures as underlying environments, but not both. In contrast, the *EcliPSe* toolkit for replication based simulation can be adapted to shared memory architectures (e.g., Sequent), distributed memory architectures (e.g., Intel i860, local networks of workstations, wide area networks) as well as hybrid environments (e.g., a combination of iPSC i860, local networks of workstations and Sequents) and performs well on each environment. Except for [28], we are unaware of any distributed simulation research based on the use of SIMD machines. Our ongoing research suggests that replication based simulation can perform favorably on SIMD machines [41] as well.

# 3 Issues motivating design

We began our research in concurrent stochastic simulation strongly motivated by the need to model certain problems in the average case analysis of algorithms and the performance of communication systems. Though research in multiprocessor based simulation has been going on for roughly a decade, we are not aware of a portable tool based on our concurrency approach. As mentioned in the previous section, the currently available tools are chiefly devoted to queueing network based simulations and geared towards execution on specific architectures.

Originating from our need for a flexible tool, our research investigations into concurrent simulation strategies were triggered by a very basic question. As researchers who are

(a) interested in using stochastic simulation for some application, and

(b) have access to an arbitrary and flexible multiprocessor configuration,

how do we go about getting a fast, tunable, and efficient model to execute on the multiprocessor system without having to devote time and effort to parallel programming issues such as synchronization, deadlock, distributed termination, etc. ?

We attempted to respond to the question with a host of designs for multiprocessor simulation systems, some of which we experimented with by coding parallel programs on a 128-node Intel i860 hypercube. The amount of effort that went into developing a working piece of parallel simulation code was of the order of 30-80 hours; changing code to accommodate design variations required several additional hours. In contrast, we found that after we had developed an *EcliPSe* prototype, we could get a concurrent stochastic simulation running within at most an hour of coding effort, given the original uniprocessor simulation

program. We are thus well aware of the frustration experienced by an analyst who is solely interested in model results and has little time to spend on learning to program specific multiprocessors or distributed systems of networked processors.

The design of *EcliPSe* began with the following four simple rules:

1. A user is required to write only a *sequential program.* This program is to be concurrentized by *EcliPSe*.

2. The emphasis is to be on replicating identical models across processors. The reasoning behind this is that $N$ processors can independently compute samples up to $N$ times faster than a single processor can.

3. A model that is too large for one processor is to be distributed across as small a number of processors as possible. The resulting distributed model is to be replicated.

4. Statistics from each processor or set of processors executing a model must be combined carefully, to eliminate problems of bias and maximize potential for speedup.

From a parallel processing view, the first point has three important benefits. The sequential model of computation reduces model development effort, simplifies task partitioning, and makes easy the job of automating the structured task interactions. The second point differentiates *EcliPSe* from other multiprocessing simulation software, since it emphasizes replication rather than distribution. The third point brings us closer to distributed simulation research though minimal distribution is advocated whenever model distribution is unavoidable due to problem size. The last point is unique to *EcliPSe* since distributed simulation software systems focus on executing a single sample path of a stochastic process at a time, whereas *EcliPSe* simulates several sample paths concurrently. It is our experience [46] that the above four conditions are sufficient to give the research community an easy-to-use and powerful tool for solving a variety of simulation problems.

# 4 The *EcliPSe* Toolkit

The *EcliPSe* toolkit enables straightforward development of stochastic simulation applications that may execute on a variety of concurrent environments. To use the toolkit, an applications developer first specifies the basic control structure of the simulation using utilities and programming abstractions provided by the *EcliPSe* system. Special data structures and simulation dependent parameters are also described in this phase of the development process. This specification, which takes the form of a procedural language description, is pre-processed by an *EcliPSe* translator. The translation process involves the incorporation

of architecture specific mechanisms and constructs for specific target concurrent environments, and produces source language code that may be directly compiled. Finally, this code is bound to appropriate *EcliPSe* libraries, which contain a variety of random number generators, termination detection mechanisms, statistics combining routines, and auxiliary support for graphical interaction and display.

The three-phase approach of specification, translation, and run time libraries adopted by the *EcliPSe* system has proven to be highly effective in concurrentizing a number of stochastic simulation applications. In particular, users are able to achieve excellent speedup levels, with the barest minimum of effort. The flexibility provided by *EcliPSe*, both in terms of machine independence as well as parallel computing paradigms, is very valuable, as are the added advantages of graphical interfaces and resilience to failures in some environments. A detailed description of the design features and implementation strategies of the system may be found in [46] ; in this section, we present a brief overview of the toolkit and highlight some of the salient facilities supported by the toolkit.

## 4.1 Parallelism Transparency

One of the primary design goals in *EcliPSe* was to enable users to specify simulation applications using a completely sequential description. This eliminates the burden of selecting an appropriate parallel computing model, dealing with synchronization and communication details, and the associated tasks of parallel coding and debugging. Such a strategy is possible due (in part) to the philosophy of maximal data parallelism and minimal functional decomposition, and to the inherent structure of stochastic simulation applications. In the simplest case, *EcliPSe* essentially performs automatic data partitioning based on the notion of multiple random number streams. The user thus describes a simulation in terms of a single random number stream, which is transparently partitioned by the translator and run time libraries to enable independent, concurrent executions on multiple processing elements. Effective statistics combination routines are used to coalesce the partial results from each of these independent computations, thereby achieving transparent concurrency.

For example, one way of achieving independent, concurrent executions is by predefining random number seed headers for different random number streams. Separating these seed headers by random number cycles of length at least 200 million (or some sufficiently large number) would allow a variety of models to execute independently without the danger of using corrupted random numbers. More general and more elaborate schemes for obtaining independence between random numbers are possible. It must be pointed out that independence between executions is not a requirement and indeed, in some instances, must be violated. One example of this occurs when processors need to exchange information (e.g., in finding the best solution through a series of searches on disjoint domains), and another

example involves inducing negative correlations between the different sampling processors in order to reduce the variance of the final estimate.

The strategy described above works well for a large number of simulation applications. However, there may exist some applications that do not fit well into such a framework; the toolkit contains provisions to cater to these applications. The toolkit allows the specification of a simulation at different levels of abstraction, with the highest level corresponding to a purely sequential specification. At lower levels, the existence of multiple processes in a concurrent simulation application become visible to the developer. Thus, an end-user may explicitly partition an application into different modules, assign specific subtasks or random number streams of a simulation to each, and even control interaction between modules. At the lowest level, applications may be directly written in (concurrent) machine dependent terms, while using the random number generators, statistics combination, termination detection, and graphical interface libraries provided by *EcliPSe*. By designing an application interface that is flexible when necessary, the toolkit enables parallelism transparency while retaining the ability to handle special requirements.

## 4.2 Application Program Structure

In the *EcliPSe* system, an application consists of a *driver* or controlling program within which the primary control structure of the simulation and a few key parameters are described. The generic driver consists of procedural language control flow statements, declarations for certain *EcliPSe* related and application related data structures, and constructs that encapsulate concurrency and simulation related activities. The generic driver is modeled after the typical structure of most stochastic simulation applications, i.e. repeated sample generation followed by statistics combination and tests for termination such as by using confidence intervals. To construct a driver, the user modifies a standard template by providing the names of application specific sample generators, combination and termination routines, and parameters such as random number stream cycle lengths, computation grainsize, and seed headers. In several cases, generation of a driver may be accomplished through the use of an interactive tool supported by *EcliPSe*, although manual editing may be necessary when an application does not conform closely to the standard template. The driver specification for a stochastic simulation to perform multidimensional integration using the sample mean method is shown below:

```
            double alpha, prec, lthresh, uthresh, grainsz, ...
/* Built-in EcliPSe declarations */
            double cur[3], dseed[2], ...
/* Application specific declarations */
```

```
                setoptions(AUTOPROCS,10,3,0)
/* EcliPSe: indicates run time determination of number of processes,
   combine every 10 samples, each sample contains 3 values etc. */

                inputdata(FROMFILE,"infile1", "%f %f\n%d %d\n %d", prec, alpha, ...)
/* EcliPSe: specifies file and format for simulation input data */

                inputcontrol(FROMFILE, "infile2", SEEDHEADS, dseed)
/* EcliPSe: file with initial seed values for many streams */

                term=0; totsample=0; for (i=0;i<3;i++) cur[i] = 0.0;
/* Application dependent initialization */

                simulate  {
                        samplegen(getintegral(dseed, totsample, cur, ...))
/* EcliPSe: specifies sample generator and arguments */
                        combine(genericcombine(cur,totsample))
/* EcliPSe: specifies statistics combination routine */
                        graphics(PLOTSAMPLE, PLOTSAMPLERATE)
/* EcliPSe: built-in plot of sample value and generation rate */
                        termcheck(confid(cur,left,right,...))
/* EcliPSe: termination check by confidence interval method */
                }

                if ((term > 0) && (totsample < uthresh)) {
/* User specified condition */
                    report(NORMAL, "Est. of integral = %f variance = %f",mean,var) }
/* Report results */
                else {
                    report(ABNORMAL)
/* Report abnormal termination */
                    }
                terminate()
/* EcliPSe: epilog activities */
```

The driver shown above is an example of an *EcliPSe* simulation application at the highest level of abstraction supported by the toolkit. In the simplest applications, a "central monitor" model may be used. Here multiple sample generator processes produce simulation samples or sample paths, while a single monitor process collects and combines them. The number and location of the multiple sample generators are determined by the system during the translation process and at execution time, and may optionally be influenced by user input during either phase.

Several other concurrent models are supported by the *EcliPSe* system. Almost all details regarding the control structure of the simulation are specified within the driver by using

appropriate *EcliPSe* constructs and/or appropriate arguments to these constructs. Some examples of the other paradigms supported are:

- Replicated monitor model, where multiple monitors collect statistics and compute results; this increases fault tolerance for long running simulations.
- Multiple monitor model, where different monitors compute different statistics based on the same or different sets of samples.
- Minimal distribution model. In this paradigm, *EcliPSe* provides for minimal functional decomposition, by permitting a small number of drivers to be specified for one application, where each driver corresponds to a functionally decomposed portion of the application. Facilities are provided for interaction and for the exchange of samples and statistics between functional modules. Each set of functional modules is replicated and executed concurrently where possible.

Almost all the effort required to use the toolkit is therefore localized to the driver generation phase, which is in itself straightforward. Sample generation routines are supplied by the user; existing sequential routines may be directly used by adding a single *EcliPSe* construct. In several cases, *EcliPSe* provided sample generators may be used, or application specific generators may be constructed by assembling modules from the toolkit repertory. Similarly, statistics combination routines, random number generators, and termination detection routines may be selected from the toolkit, supplied by the user, or both.

## 4.3 Translation and Execution

The driver and other source level components of a simulation application are preprocessed by the translator in a manner dependent upon the targeted execution environment. The execution environments currently supported are uniprocessors, loosely coupled heterogeneous networks of scalar machines, distributed memory multiprocessors, and shared memory parallel machines. During the translation phase, the *EcliPSe* constructs are converted to reflect the computing model supported by the target hardware platform, without introducing machine or operating system specifics. For example, message transmission and reception abstractions are substituted when the target environment is a loosely coupled network or a distributed memory multiprocessor. For loosely coupled networks, predefined modules are incorporated that permit continued execution when processing elements or interconnection links fail. In addition, environment dependent *EcliPSe* initialization routines and source code for certain run-time actions are added. The output of the translation phase consists of a collection of compile-ready source code files. If desired, options that will take effect during

execution may be specified during translation; examples include upper and lower limits on the number of concurrent processes to be used, the location and formats of simulation input data, and graphical interface routines for interactive monitoring of the simulation.

The source level outputs of the translation phase are compiled for specific machines and linked against architecture-dependent, *EcliPSe* provided, libraries. During this phase, communication and synchronization abstractions in the preprocessed code are bound to machine dependent system calls and library routines. Certain other housekeeping functions, such as shared address space management routines for shared memory multiprocessors, are also included during the compilation and linking phase. The simulation object code that is produced may directly be executed. During execution, *EcliPSe* inputs as well as application inputs may be necessary, depending upon the options specified in the driver. Examples of the former are the number of processes to be used, the number of replicated monitors required, and the grainsize (i.e., the number of samples to be generated before statistic-combination is performed). Examples of simulation application input include initial seed values, and precision and confidence interval ranges.

## 4.4 Run Time Environment

As explained briefly earlier, concurrency in *EcliPSe* is achieved primarily by parallel processing of separate seed streams. The run time routines place multiple sample generator processes on different processing elements; and samples are routed by *EcliPSe* libraries to the appropriate monitor process(es) based upon the model in use. For the functionally decomposed model, *EcliPSe* routines set up interaction channels between different functional modules, as dictated by the distribution structure specified in the driver and other modules. Monitor processes collect statistics, combine samples, and check whether simulation termination conditions have been met. In addition, auxiliary routines assist in other tasks such as monitoring load imbalances or processor failures, managing memory and secondary storage space containing sample values, accepting certain forms of interactive user input, and displaying graphical information regarding the status of the simulation.

The graphical interface component of the toolkit is a very valuable facility. These graphics routines provide the ability to monitor a simulation run while in progress, and based upon this a user may decide to alter the run or provide additional input. *EcliPSe* supports several built-in graphics displays, primarily consisting of two-dimensional plots of sample values, generation rates, convergence behavior, and load balance. These rotuines are automatically invoked and execute while the simulation is in progress, provided that such a display option was selected in the driver specification. In addition, *EcliPSe* constructs are provided whereby user-written portions of the simulation may periodically feed values to graphics routines that display the information as the simulation proceeds. At present, only

simple plotting facilities are supported, but enhancements are under development. This facility also permits the user to define conditions under which simulation parameters are modified at run time based on user input. For example, the application may specify the need for fresh input if termination is not achieved with one million samples. When this condition is detected, the system requests interactive user-input for various parameters, (e.g., a wider confidence interval, additional or fewer processing elements, larger grainsize).

### 4.5 Status and Ongoing Work

The toolkit has been implemented on a variety of machines including the Intel iPSC/860 and iPSC/2 hypercubes, Sequent shared memory multiprocessors, IBM RIOS/6000 workstations, Sun servers and workstations, and heterogeneous networks of the above. From our experiences with these implementations, it appears that porting the toolkit to other architectures will be straightforward. In our experiments as well as during use by others, we have encountered few difficulties either with the use of *EcliPSe* or its performance and overheads. Indeed the versatility provided by the system, its ease of use, and good concurrent performance are significant and valuable for a variety of stochastic simulation applications.

Several aspects of the toolkit however, require further research and development, and some of these enhancements are ongoing while others are planned for the future. One area that is only minimally exploited by *EcliPSe* currently is the semiautomatic generation of the driver and other parts of the simulation specification. We are working on an interactive graphical tool that will permit the assembly of a simulation specification with little or no coding required of the user. Along the same lines, the run-time graphical displays and interaction environment are being enhanced to permit greater flexibility and generality. From the simulation viewpoint, our efforts will be concentrated on more natural specification and development for functionally decomposed distributed simulation.

## 5 Empirical Results

We report some timing results on a suite of eight simulation experiments which were performed using the *EcliPSe* toolkit. Each of these experiments was independently conducted in the various execution environments described in the previous subsection, and only makes use of the simple replication feature of *EcliPSe*. That is, in each experiment it was possible to fit the entire model on a processor and replicate models across a set of available processors. In addition, though multiple monitors are allowed by *EcliPSe*, a single monitor is used in these experiments.

Given a model $\mathcal{M}$ that is to be executed on an environment with $N$ processors, *EcliPSe* replicates $\mathcal{M}$ on each of the $N$ processors, ensuring that each processor $j$ utilizes a specific

random number stream $\mathcal{R}_j$ for the replica $\mathcal{M}_j$ that it hosts, for $0 \leq j \leq N-1$. In general, replica $\mathcal{M}_j$ on processor $j$ generates statistics that are independent of statistics generated on processor $k$, for $j \neq k$, although variations of this random number stream allocation for the purposes of variance reduction or comparison of multiple alternatives are allowed. A unique processor, say processor 0, is arbitrarily selected to perform the monitor function, while each processor $j$ executes replica $\mathcal{M}j$, for $1 \leq j \leq N-1$. While the monitor may or may not execute replica $\mathcal{M}_0$, depending on the strategy used, it is the monitor's responsibility to receive incoming statistical data from the executing processors and pool these data together to create a single set of statistics. In the following experiments, the monitor was used solely for the purpose of pooling statistics and not for executing replica $\mathcal{M}_0$, except in the case $N = 1$, where the monitor was forced to perform both functions. Since monitoring eliminates one processor from the sampling pool, the speedup obtained when $m$ processors are used is always bounded from above by $m-1$, for $1 < m \leq N$.

## 5.1 Pooling statistics

During its execution, each replica $\mathcal{M}_j$ generates a running vector or matrix of statistics which is application dependent. For example, using regenerative simulation [7] for a network of $K$ queues, the vector would contain running estimates of the first and second moments of the number of customers served and the total waiting time in each regenerative cycle seen at each queue. At appropriate, user-defined times controlled by a "grainsize" parameter, the current statistics vector is reported to the monitor, which then proceeds to combine this data with data already received from other processors. Each time it obtains an estimate, the monitor calls a user-defined termination procedure, for example a confidence interval building routine, in order to determine if a termination condition has been met. The process continues until the termination condition is satisfied, and the monitor terminates the concurrent simulation.

There are a variety of techniques that can be used by the monitor for combining statistics sent from the different executing processors. These techniques are dependent on the properties of estimators used by the executing model and precisely what one expects of the estimate constructed by the monitor. For example, one method (e.g., [21]) for constructing an unbiased estimate is to take an equal number of samples from each processor in constructing the final estimate. The naive strategy of computing the final estimate using an arbitrary number of samples from each processor is known to produce a biased estimate. In the following experiments the monitor pools statistics by combining an equal number of samples from each processor. It should be clear that this statistic-combining method can be improved for various models, so that the timings reported here are to some extent pessimistic; they are determined by the "slowest" processor to complete the estimate that is used by the monitor to effect termination.

A set of at least five parameters is used to control each of the following experiments. Each model $\mathcal{M}_j$, executing on processor $j$, is required to compute a minimum of *lt* (lower threshold) samples before reporting the current estimate vector or matrix to the simulation monitor. The interval of computation between consecutive reports is controlled by a grain-size parameter called *gsize*. This parameter controls the overall amount of communication, usually being assigned large values for models with high sampling rates, and smaller values for models with low sampling rates. If a user is unable to define an appropriate value of *gsize*, *EcliPSe* is capable of selecting and dynamically adjusting this parameter to control load on the monitor and optimize performance. A *ut* (upper threshold) parameter defines an upper bound on the total number of samples generated before abnormal model termination. Under normal conditions, the monitor begins to call a user-defined termination routine (or one of several confidence interval building routines in the *EcliPSe* library) after each sampling process has reported in *lt* samples, and thereafter in multiples of *gsize* samples. This enables the monitor to test if a specified termination requirement has been met. In each of the following experiments model termination is achieved using a $(1-\alpha)\%$ confidence interval based on the Student t-distribution. The half-width of the interval relative to the absolute value of the quantity being estimated is controlled by a precision parameter $\gamma$, $0 < \gamma < 1$, where smaller values of $\gamma$ result in larger execution times.

## 5.2 Interpretation of results

Following a description of each experiment, a pair of tables containing timing information for the experiment on various architectures and varying numbers of processors is shown. For each machine environment displayed in the tables, the first row of numbers represents the time (in cpu seconds) required to execute the model with normal model termination, and the second row of numbers represents the total number of samples generated. For the latter, a "k" or an "m" alongside any number denotes sample-size in units of a thousand, or a million, respectively.

In each of the following experiments timing measurements are made on the monitor, essentially capturing the amount of time elapsed from the start of the run until termination. The reported measurements are accurate on the hypercubes, where processors devote all their attention to the simulation application. On the other architectures, transient system load (due to other applications) on sampling processors during the simulation increases the timing measured on the monitor, making our measurements conservative.

The speedup obtained for a given model when using $n$ processors can be computed as the ratio of the time taken by a single processor and the time taken by $n$ processors to run the model. When each of *EcliPSe*'s instances executes on a different processor, simulation speedup will be bounded from above by $(n-1)$, because one processor will have to dedicate itself to the task of monitoring the simulation experiment. This situation arises, for example,

when *EcliPSe* runs on the hypercubes, but not on the other architectures where the monitor behaves as just another process. This explains why speedup is roughly unity when $n = 2$ for the hypercube based experiments in the tables given below. When $n = 1$, *EcliPSe* forces the single executing instance to perform both functions, that of sampling as well as monitoring the simulation run.

The first table shown after the description of each experiment exhibits timings on a 128-node Intel i860, where each node supports up to eight Mb of memory, and a 64-node iPSC/2 with nodes supporting one Mb of memory each. The second table in each experiment exhibits timings on an 8-processor Sequent Symmetry (at Purdue University), a local network of eight IBM RIOS workstations (RIOS LN, at Oak Ridge), a local network of eight SUN4 workstations (SUN4 LN, at Emory University), and finally a heterogeneous wide-area network (HWAN). The machines making up the various sized HWAN configurations are explained in the table, where the notation S(Sun), R(Rios), Q(Sequent), E(Emory University), P(Purdue University), and O(Oak Ridge National Laboratory) is self-explanatory. Table (1b) defines defines the makeup of the HWAN environment as the number of processors is varied.

The timings exhibited for the HWAN experiments deserve some explanation. Depending on the number of processors used, the heterogeneous machines defining the HWAN environment varied, as mentioned in the previous paragraph. Since the monitor waits for an equal number of samples from each executing model before computing a new estimate, slower executing processors are certain to slow down the entire simulation process. These slower processors either operate at a slower rate (e.g., a processor on the Sequent Symmetry is slower than a SUN4, which in turn is slower than an IBM RIOS processor), or are loaded with some other application which causes them to report statistics to the monitor at a delayed pace. This phenomenon is clearly seen, for example, in the second, third, and fourth columns of each table containing timings for the HWAN configuration.

The timings shown in the tables suggest that there is a discrepancy between speedup results for some experiments on the two hypercubes (e.g., Experiment 3). This is due to the fact that either the problem sizes, or some parameter values in the model or the simulation environment are different when the model is run on the hypercubes. For example, in the case of Experiment 3, the problem size on the iPSC/2 is smaller than on the i860 (i.e., 500 states versus 1000 states), and the *grainsize* parameter is larger on the iPSC/2 than on the i860 (i.e., 10 samples per combination versus 1 sample per combination). The larger problem size ensures that there is a much larger variation in the sample generation rate on the i860 in comparison to the iPSC/2. Using *grainsize* = 1 on the i860 forces every processor to report each new sample it obtains to the monitor. In combination, these factors increase communication overhead and effectively reduce the rate at which the monitor operates. For $n \geq 64$, the results indicate that the effects are sufficient to limit estimation rate, causing the speedup for iPSC/860 to drop considerably.

**Experiment 1:** *Multidimensional Integration*

It is well known that the problem of computing integrals in higher dimensions can require a tremendous amount of calculation. Monte Carlo methods are known [14] to be more efficient than analytical techniques when the number of dimensions is beyond seven.

A classical Monte Carlo technique for estimating multidimensional integrals is the *sample-mean* method. For ease of explanation, we assume the function $h(\mathbf{x})$ to be integrated is bounded and non-negative over domain $\mathbf{R}_h$ of vector $\mathbf{x}$ in a $(d-1)$ dimensional space. In order to estimate

$$\mathbf{I} = \int_{\mathbf{R}_h} h(\mathbf{x}) d\mathbf{x} \tag{1}$$

we begin by choosing a density function $f(\mathbf{x})$ defined over $\mathbf{R}_h$. Then $\mathbf{I}$ can be expressed as

$$\mathbf{I} = \int_{\mathbf{R}_h} \left[\frac{h(\mathbf{x})}{f(\mathbf{x})}\right] f(\mathbf{x}) d\mathbf{x} = E\left[\frac{h(\mathbf{X})}{f(\mathbf{X})}\right] \tag{2}$$

where $\mathbf{X}$ is a random vector whose density is $f(\cdot)$, and $E[\cdot]$ denotes expectation.

A sampling process selects a certain number $n$ of points $\{\mathbf{x}^{(j)};\ 1 \leq j \leq n\}$ randomly from $\mathbf{R}_h$, according to the density $f(\cdot)$. An estimate $\hat{\mathbf{I}}$ of $\mathbf{I}$ is thus obtained as the sample mean of $n$ observations of $h(\cdot)$, where

$$\hat{\mathbf{I}} = \frac{1}{n} \sum_{j=1}^{n} \frac{h(\mathbf{x}^{(j)})}{f(\mathbf{x}^{(j)})} \tag{3}$$

In this experiment we constructed an estimate of a multi-dimensional integral of an exponential in the positive quadrant. The performance results for this experiment are shown in Tables 1a and 1b. For all the environments, parameters were fixed at $lt = 100$, $ut = 5 \times 10^6$, $gsize = 5000$, $\alpha = 10^{-4}$ and $\gamma = 10^{-3}$.

As explained in the previous subsection, the apparently haphazard timings for the HWAN experiment are strictly a result of the statistic-combining strategy and the varying speeds of processors being used in defining the HWAN. Timings will improve dramatically through an improvement in combining strategy. The haphazard nature of the timings can be changed to yield numbers which decrease, as the number of processors increases, if processors are added to the HWAN in order of *nondecreasing* speeds.

**Experiment 2:** *Order Statistics*

In many Monte Carlo studies the random quantities of interest are the *extremes* of a sample. One example is the maximum (or minimum) of a set of random variables, such as in a PERT simulation, where an event is realized only after all preceding activities are

Hypercubes

| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|
| RX | 745 | 770 | 275 | 118 | 56 | 27 | 13 | 7 |
| | 27.35m | 27.35m | 27.42m | 27.15m | 27.24m | 27.17m | 27.32m | 27.44m |
| IPSC | 12521 | 12726 | 4252 | 1811 | 851 | 411 | 202 | NA |
| | 27.3m | 27.3m | 27.5m | 27.54m | 27.2m | 27.32m | 27.44m | NA |

Time/Sample-size for Integral estimation (Intel Hypercubes)

**Table (1a)**

Other

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| SEQUENT | 6419 | 3213 | 2141 | 1615 | 1286 | 1073 | 921 |
| | 27.35m | 27.44m | 27.15m | 27.32m | 27.25m | 27.32m | 27.8m |
| RIOS LN | 531 | 266 | 180 | 138 | 107 | 91 | 79 |
| | 27.92m | 27.35m | 27.35m | 27.42m | 27.28m | 27.35m | 27.42m |
| SUN4 LN | 1492 | 783 | 512 | 392 | 346 | 292 | 265 |
| | 27.31m | 27.40m | 27.36m | 27.35m | 27.39m | 27.8m | 27.61m |
| HWAN | 1498 | 620 | 2017 | 1496 | 1149 | 962 | 780 |
| | 27.34m | 27.6m | 27.39m | 27.2m | 27.15m | 27.18m | 27.24m |
| | 2S(E) | 2S(E) + R(O) | 2S(E) + R(O) + Q(P) | 2S(E) + 2R(O) + Q(P) | 3S(E) + 2R(O) + Q(P) | 4S(E) + 2R(O) + Q(P) | 4S(E) + 3R(O) + Q(P) |

Time/Sample-size for Integral estimation (Sequent, Local-Nets, Wide-area Nets)

**Table (1b)**

Hypercubes

| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|
| RX | 462 | 465 | 189 | 81 | 39 | 19 | 10 | 4 |
| | 1.72m | 1.72m | 1.75m | 1.79m | 1.72m | 1.79m | 1.66m | 1.52m |
| IPSC | 6797 | 6841 | 2844 | 1226 | 565 | 296 | 153 | NA |
| | 1.72m | 1.72m | 1.75m | 1.73m | 1.72m | 1.79m | 1.05m | NA |

Time/Sample-size for Order-statistic estimation (Intel Hypercubes)
**Table (2a)**

complete. In this experiment, we use a selective inversion technique to generate samples and estimate the expected value of the maximum of a set of independent, possibly non-identically distributed random variables.

Let $X_i$ be a random variable with cumulative distribution function (cdf) $F_i(\cdot)$, $1 \leq i \leq m$, and assume that the $X_i$'s are independent. We are interested in obtaining

$$E[M] = E[\max\{X_1, X_2, \ldots, X_m\}] \tag{4}$$

which is the expected value of the maximum order statistic of this set of random variables. In general, computing $E[M]$ analytically is a difficult proposition indeed for large $m$. A simple Monte Carlo approach to obtaining an estimate $\hat{E}[M]$ of $E[M]$ would be to repeatedly generate samples $x_1^{(j)}, \ldots, x_m^{(j)}$ of the random variables $X_1, \ldots, X_m$, for each $j$, up to some number $n > 0$. Our required estimated would be given by

$$\hat{E}[M] = \frac{1}{n} \sum_{j=1}^{n} x_{\max}^{(j)} \tag{5}$$

where $x_{\max}^{(j)} = \max\{x_1^{(j)}, \ldots, x_m^{(j)}\}$. The direct approach to obtaining $x_{\max}^{(j)}$ is an $O(m)$ operation, with $m$ steps required for generating the quantities $x_1^{(j)}, \ldots, x_m^{(j)}$, and some number of steps required for finding the maximum.

In this experiment, we use a technique developed by Schmeiser [45] to generate values of $x_{\max}^{(j)}$ at a cost *less* than $O(m)$. Assuming that the values of the $X_i$'s are generated using the inverse transformation technique

Other

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| SEQUENT | 5165 | 2614 | 1796 | 1342 | 1104 | 895 | 792 |
| | 1.72m | 1.73m | 1.74m | 1.73m | 1.74m | 1.73m | 1.79m |
| RIOS LN | 475 | 249 | 168 | 130 | 110 | 94 | 82 |
| | 1.71m | 1.73m | 1.78m | 1.74m | 1.73m | 1.73m | 1.74m |
| SUN4 LN | 1331 | 685 | 472 | 346 | 295 | 247 | 210 |
| | 1.74m | 1.76m | 1.75m | 1.78m | 1.74m | 1.74m | 1.73m |
| HWAN | 1342 | 684 | 1822 | 1239 | 982 | 798 | 693 |
| | 1.73m | 1.73m | 1.74m | 1.73m | 1.75m | 1.74m | 1.73m |

Time/Sample-size for Order-statistic estimation (Sequent, LNs, HWANs)

**Table (2b)**

$$x_i^{(j)} = F_i^{-1}(u) \tag{6}$$

where $u$ is a uniform (0,1) random value (as is the case for the Weibull, Poisson, binomial, geometric, and arbitrary histograms [13] ), Schmeiser's method requires partitioning the (0,1) interval into $m$ pieces. For each $j$, the quantity $x_{\max}^{(j)}$ is obtained by generating only *those* $x_i^{(j)}$, $1 \leq i \leq m$, that are likely to be candidates for $x_{\max}^{(j)}$. In this way significant savings in computational effort can be had, especially for large $m$.

In this experiment we estimate the expected value of the maximum of a set of $m$ Weibull random variables, for $m = 20$. Increasing $m$ increases the timings but improves the speedups. The performance results for this experiment are shown in Tables 2a and 2b. Except for *gsize*, parameters for all environments were fixed at $lt = 100$, $ut = 3 \times 10^6$, $\alpha = 10^{-4}$, and $\gamma = 10^{-3}$. For the i860 and RIOS environments, $gsize = 6000$, while for the others, $gsize = 5000$.

**Experiment 3:** *Hitting-Times in Markov chains*

Markov chains are known to be useful in a variety of modeling contexts, such as the analysis of algorithms [40], system reliability [47], queueing applications [34] and Monte

Hypercubes

| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|
| RX | 2407 | 2486 | 1001 | 553 | 290 | 150 | 91 | 57 |
| 1000×1000 | 4224 | 4224 | 4263 | 4295 | 4301 | 4278 | 4215 | 4191 |
| IPSC | 270 | 277 | 93 | 41 | 22 | 14 | 6 | NA |
| 500×500 | 4490 | 4490 | 4410 | 4410 | 4500 | 4650 | 4580 | NA |

Time/Sample-size for Hitting-time estimation (Intel Hypercubes)

**Table (3a)**

Carlo based optimization [44]. In this experiment, we focus on a discrete time Markov chain $\{Y_m; m \geq 0\}$ operating on a finite state space $\mathbf{S} = \{0, 1, 2, 3, \ldots, k\}$.

Let $\{Y_m; m \geq 0\}$ make time-homogeneous transitions according to a given transition probability matrix $\mathbf{P} = [p_{i,j}]$. For a subset of states $A \subset \mathbf{S}$, the first hitting time $T_A$ is defined as

$$T_A = \min\{j \geq 0; Y_j \in A\} \tag{7}$$

Using $E_r(\cdot)$ to denote expectation conditional on $Y_0 = r$, assuming a positive recurrent chain means $E_r(T_A) < \infty$, and the expected hitting-times can be determined by solving a system

$$E_i(T_A) = \begin{cases} 1 + \sum_j P(i,j) E_j(T_A) & i \notin A \\ 0 & i \in A \end{cases} \tag{8}$$

for fixed $A \subset \mathbf{S}$. An alternate view of the random variable $T_A$, known as a *phase-type* random variable [34], is as the time to absorption in a Markov chain. If the states in $A$ are all lumped together into a single *absorbing* state, then $E_r(T_A)$ becomes the expected time to absorption for the chain, given the initial state is $Y_0 = r$. In either case, obtaining $E_r(T_A)$ requires the inversion of the matrix $(\mathbf{I} - \mathbf{Q})$, where $\mathbf{Q}$ is a special submatrix of $\mathbf{P}$. Clearly, when $k$ is large (say of the order of $10^3$ or $10^6$), obtaining the *fundamental* matrix $(\mathbf{I} - \mathbf{Q})^{-1}$ of $\mathbf{Q}$ is computationally prohibitive via direct computation. Alternatively, using $r$ as the initial state for the chain $\{Y_m\}$, we can simulate a sequence of independent realizations $y_1, y_2, \ldots, y_n$ of the hitting-time. That is, $y_j$ is defined as the number of steps required to

Other

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| SEQUENT | 239 | 132 | 95 | 69 | 55 | 45 | 39 |
| 500×500 | 4490 | 4330 | 4510 | 4420 | 4350 | 4290 | 4460 |
| RIOS LN | 5910 | 3392 | 2315 | 1715 | 1388 | 1371 | 1146 |
| 1000×1000 | 4360 | 4240 | 4140 | 4190 | 4230 | 4360 | 4330 |
| SUN4 LN | 53 | 27 | 19 | 14 | 11 | 9 | 8 |
| 500×500 | 4420 | 4350 | 4380 | 4410 | 4510 | 4420 | 4530 |
| HWAN | 56 | 32 | 78 | 55 | 48 | 36 | 29 |
| 500×500 | 4420 | 4490 | 4500 | 4320 | 4350 | 4300 | 4410 |

Time/Sample-size for Hitting-time estimation (Sequent, LNs, HWANs)

**Table (3b)**

take the chain from state $r$ to the set $A$ on the $j$th attempt. In this way, we can construct the estimate

$$\hat{E}_r(T_A) = \frac{1}{n} \sum_{j=1}^{n} y_j \tag{9}$$

of the expected hitting-time $E_r(T_A)$.

In this experiment we estimate the expected hitting-time of a Markov chain with $k+1$ states, where $k = 1000$ for the i860 and RIOS environments, and $k = 500$ for the others. In each case the Markov chain was started in state $k$, and forced to terminate in absorbing state 0. Both situations required roughly 4200 samples for model termination. The performance results for this experiment are shown Tables 3a and 3b. For the i860 and RIOS environments, $lt = 30$, $ut = 10^5$, $gsize = 1$, $\alpha = 10^{-3}$, and $\gamma = 0.05$. For the others, $lt = 50$, $ut = 10^5$, $gsize = 10$, $\alpha = 10^{-3}$, and $\gamma = 0.05$.

**Experiment 4:** *Dijkstra's Self-Stabilization Algorithm*

Distributed algorithms are known to pose formidable problems to analysts interested in measuring algorithmic complexity, in particular, average complexity [22]. These analytic difficulties stem largely from multidimensionality and related enumeration problems. A

good example of a distributed algorithm whose average run-time complexity can be hard to measure is the $K$-state algorithm of Dijkstra [10].

The $K$-state algorithm is one of several algorithms developed by Dijkstra [10] for effecting a *self-stabilization* mechanism for $M$ processors on a *unidirectional* ring. Each processor $j$, $1 \leq j \leq M$, initially possesses a label $\ell(j)$, $1 \leq \ell(j) \leq K$, with $K > M$. We take processor 1, called the END-processor, to be a unique processor which functions differently from the others when executing the algorithm.

Assuming that messages travel from lower numbered processors to higher numbered ones (and from processor $M$ to the END-processor), each processor $j$ examines label information it receives from its *upstream* neighbor and accordingly updates a local boolean variable $f(j)$, $1 \leq j \leq M$. For the END-processor,

$$\begin{aligned} \ell(1) = \ell(M) &\Rightarrow f(1) = 1 \\ \ell(1) \neq \ell(M) &\Rightarrow f(1) = 0 \end{aligned} \tag{10}$$

and for each processor $j$, $2 \leq j \leq M$,

$$\begin{aligned} \ell(j) = \ell(j-1) &\Rightarrow f(j) = 0 \\ \ell(j) \neq \ell(j-1) &\Rightarrow f(j) = 1 \end{aligned} \tag{11}$$

so that at any given instant, the boolean count is given by

$$c = \sum_{j=1}^{M} f(j) \tag{12}$$

where $0 < c \leq M$. If $c > 1$, the distributed system is said to be in an *unstable* state. Whenever the system enters an unstable state, the $K$-state algorithm brings the system back into a stable state by defining asynchronous processor actions as follows. For the END-processor,

$$\textbf{if } (f(1) = 1) \textbf{ then } \ell(1) = \ell(1) \textbf{ mod } K + 1 \tag{13}$$

and for each processor $j$, $2 \leq j \leq M$,

$$\textbf{if } (f(j) = 1) \textbf{ then } \ell(j) = \ell(j-1) \tag{14}$$

It is easy to see that when $c = 1$, the system continues to operate in the stable state indefinitely. However, should some erroneous condition arise (e.g., message error or processor malfunction) the $K$-state algorithm takes over immediately to bring the system back into

Hypercubes

| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|
| RX | 635 | 639 | 216 | 99 | 42 | 21 | 10 | 5 |
| | 840 | 840 | 840 | 882 | 830 | 804 | 756 | 762 |
| IPSC | 9830 | 9835 | 3286 | 1477 | 646 | 321 | 157 | NA |
| | 850 | 850 | 860 | 842 | 882 | 810 | 844 | NA |

Time/Sample-size for Dijkstra's algorithm (Intel Hypercubes)
**Table (4a)**

a stable state. Our interest is in determining the *average* number of steps required for self-stabilization as a function of $K$ and $M$, given an initial boolean count of $c > 1$.

In this experiment we estimate the expected run-time of Dijkstra's $K$-state algorithm for a system with $M = 1000$ processors and $K = 6000$. The algorithm is made to reduce the boolean count $c$, initially set at $c = 100$, to $c = 1$. The performance results for this experiment are shown in Tables 4a and 4b. For all the environments, the parameters were fixed at $lt = 4$, $ut = 10^6$, $gsize = 2$, $\alpha = 0.05$, and $\gamma = 0.01$.

**Experiment 5:** *Estimating Unknowns in a Linear System*

There are a variety of deterministic methods for solving systems of the form

$$A\mathbf{x} = \mathbf{b} \tag{15}$$

where $A$ is a given order $n$ matrix, and $\mathbf{b}$ is a given order $n$ vector. Considerable attention has been given to extending these methods for efficient execution in multiprocessor environments (e.g., [20]). From the viewpoint of analysts interested in solving very large, dense systems, these deterministic methods all suffer from an inherent drawback in that there is a limit to the extent of possible decoupling between subtasks in their algorithms. As a result, communication between subtasks executing on different processors is required. This tends to limit speedup as the number $N$ of available processors increases. Another disadvantage of deterministic methods is the intricate pattern of communication and computation which must be maintained between processors, thus entailing a high degree of complexity in programming.

In principle, it is always possible [43] to put (15) in the form

Other

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| SEQUENT | 18376 | 9215 | 6164 | 4622 | 3773 | 3086 | 2633 |
| | 890 | 890 | 872 | 844 | 858 | 878 | 898 |
| RIOS LN | 3853 | 1967 | 1297 | 998 | 807 | 763 | 703 |
| | 850 | 850 | 840 | 880 | 848 | 880 | 890 |
| SUN4 LN | 10788 | 5527 | 3709 | 2753 | 2291 | 2136 | 1996 |
| | 882 | 844 | 820 | 840 | 844 | 842 | 840 |
| HWAN | 10794 | 4368 | 14829 | 10495 | 7652 | 7028 | 5869 |
| | 820 | 842 | 844 | 836 | 842 | 840 | 850 |

Time/Sample-size for Dijkstra's algorithm (Sequent, LNs, HWANs)
**Table (4b)**

$$\mathbf{x} = C\mathbf{x} + \mathbf{b} \tag{16}$$

where $C = I - A$ and $\|C\| < 1$. Here we take $\|\cdot\|$ to be

$$\|C\| = \max_i \sum_{j=1}^{n} |c_{ij}| \tag{17}$$

In accordance with stationary linear iterative schemes for solving (15), assuming that $\mathbf{x}^{(0)} \equiv 0$ and $C^0 \equiv I$, it can be shown [23] that

$$\mathbf{x}^{(k+1)} = \sum_{r=0}^{k} C^r\mathbf{b} \tag{18}$$

and in the limit

$$\lim_{k\to\infty} \mathbf{x}^{(k)} = (I - C)^{-1}\mathbf{b} = A^{-1}\mathbf{b} = \mathbf{x} \tag{19}$$

for $A$ nonsingular. In order to estimate $[C^r\mathbf{b}]_i$, which is the $i$th-component of the vector $C^r\mathbf{b}$, we proceed as follows. We choose numbers $p_{ij}$ and $z_{ij}$, $p_{ij} \geq 0$, $\sum_j p_{ij} = 1$, such that

Hypercubes

| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|
| RX | 209 | 212 | 72 | 31 | 14 | 8 | 4 | 2 |
| 1000×1000 | 282k | 281.2k | 280.0k | 281.2k | 286.7k | 285.0k | 285.7k | 284.5k |
| IPSC | 572 | 193 | 82 | 36 | 19 | 10 | 5 | NA |
| 500×500 | 76.1k | 76.2k | 75.2k | 78.4k | 77.5k | 76.1k | 76.4k | NA |

Time/Sample-size for Linear System estimation (Intel Hypercubes)

**Table (5a)**

$c_{ij} = z_{ij}p_{ij}$, $1 \leq i, j \leq n$. That is, $[p_{ij}]$ is taken to be the transition probability matrix of a Markov chain $\{X_m;\ m \geq 0\}$ such that the random variable $X = z_{x_0,x_1} \cdot z_{x_1,x_2} \cdots\cdots z_{x_{r-1},x_r} \cdot b_{x_r}$ possesses the property

$$E[X|X_0 = i] = [C^r\mathbf{b}]_i \tag{20}$$

The conditional expectation in (20) is obtained by simulating realizations of sample paths of length $r$ in the chain $\{X_m\}$. Since sample paths of length $r$ are contained within sample paths of length $k$, for $0 \leq r \leq k$, it is clear that the $i$th component of $\mathbf{x}^{(k+1)}$ in (18) can be computed via (20) and sample paths of length $k$. Details of such a computation can be found, for example, in [43]. In order to compute all the components of $\mathbf{x}^{(k+1)}$ in (18) simultaneously, we proceed as follows. Let $\mathbf{S} = \{1, 2, \ldots, n\}$ and $\mathbf{H}_j = \{X_0, X_1, \ldots, X_j\}$ for $j \geq 0$. Defining

$$T = \min\{j | \mathbf{S} \cap \mathbf{H}_j = \mathbf{S}\} \tag{21}$$

to be the *covering time* of chain $\{X_m\}$, we sample realizations of length $(T + k)$ with initial state $X_0$ chosen randomly from $\mathbf{S}$. Given that $\{X_m\}$ is ergodic, $E[T] < \infty$ almost surely, and paths of length $k$ are constructed from a single path of length $(T + k)$ for each state in $\mathbf{S}$.

In this experiment we estimate the value of unknown $x_{100}$ in an order $n$ system, where $n = 1000$ for the i860 and RIOS environments, and $n = 500$ for the others. In both cases, the length of the covering path was set at $k = 30$. The performance results for this experiment are shown in Tables 5a and 5b. For the i860 and RIOS environments, parameters were set at $lt = 1000$, $ut = 10^6$, $gsize = 250$, and $\alpha = \gamma = 0.01$. For the others, parameters were

Other

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| SEQUENT | 880 | 451 | 298 | 219 | 178 | 151 | 128 |
| 500×500 | 76.2k | 77.4k | 77.2k | 77.4k | 76.8k | 78.1k | 76.4k |
| RIOS LN | 223 | 114 | 79 | 60 | 46 | 37 | 35 |
| 1000×1000 | 76.4k | 77.2k | 77.2k | 76.8k | 77.4k | 78.2k | 78.2k |
| SUN4 LN | 622 | 314 | 170 | 158 | 125 | 109 | 91 |
| 500×500 | 77.4k | 76.8k | 76.6k | 77.0k | 77.0k | 76.4k | 78.0k |
| HWAN | 625 | 249 | 663 | 603 | 390 | 341 | 262 |
| 500×500 | 77.4k | 77.2k | 77.8k | 77.2k | 77.2k | 77.4k | 77.0k |

Time/Sample-size for Linear System estimation (Sequent, LNs, HWANs)

**Table (5b)**

set at $lt = 1000$, $ut = 10^6$, $gsize = 100$, and $\alpha = \gamma = 0.01$. It is remarkable that one can estimate at least one unknown of an order 1000 system, to within three to four decimal places of accuracy, within two cpu seconds on an i860.

**Experiment 6:** *Tail Probabilities in Queues*

Queuing systems are known to be useful in modeling computer and communication systems [34]. However, as is often the case with random phenomena, obtaining information concerning transient or rare behavior in queues is a subject of considerable difficulty. While these phenomena present themselves in a variety of queueing systems, for convenience we will focus our attention on single server queues.

Consider a single server queue where customers are served in the order of their arrival, and interarrival times and service times are arbitrarily distributed random variables. Let $\{L_m; m \geq 0\}$ be the discrete process representing the queue size at customer departure instants. The stochastic equation governing successive queue sizes at departure instants is given by

$$L_{m+1} = (L_m - 1)^+ + A_{m+1} \tag{22}$$

where $L_m$ is the number of customers queued at the $m$th departure and $A_{m+1}$ is the number of arrivals during the service time of the $(m+1)$st customer. Assuming a stable system,

we are interested in estimating the probability $P[L > k]$, where $L$ is the stationary queue length random variable and $k$ is some nonnegative integer. Clearly, the events $\{L_m > k\}$ are *rare* events for sufficiently large $k$, and thus tend to make such simulations expensive.

There has been considerable interest in techniques for speeding up the estimation of $P[L > k]$ using large deviations and importance sampling via an optimal change of measure. These attempts have been successful (e.g., see [37]) for the M/M/1 queue, open Jackson networks, and the GI/GI/1 queue. While these techniques are undoubtedly useful and important, in the absence of a general enough framework that handles various systems, there will always be a strong motivation for using direct Monte Carlo methods and parallel simulation for obtaining information about rare events.

A good example of a queueing system that does not succumb easily to the change of measure techniques described above, is the GI/SM/1 queue, a variation of the M/SM/1 queue introduced by Neuts [35]. While customer interarrival times remain a renewal process, each customer is one of several, say $N$, types. A customer of type $j$ is followed by a customer of type $i$ with probability $p_{ij}$, so that a matrix $[p_{ij}]$ is required to describe how arrivals of different types occur, $1 \leq i, j \leq N$. Upon receiving service, a customer of type $i$ requires a service time depending on his type, $1 \leq i \leq N$.

A straightforward regenerative simulation approach to obtaining an estimate of $P[L > k]$ would be as follows. We simulate a sequence $\{C_m\}$ of *regenerative cycles*, where each regenerative cycle (see [7]) is a busy period plus the following idle period, for $m \geq 0$. Using $|E_m|$ to denote the amount of time the queue size exceeds $k$ in the $m$th regenerative cycle, we take

$$\hat{P}[L > k] = \frac{\sum_{j=1}^{r} |E_j|/r}{\sum_{j=1}^{r} |C_j|/r} \tag{23}$$

to be an unbiased estimator of $P[L > k]$ for an $r$ that satisfies the desired termination condition.

In this experiment we estimate the queue tail probability for an GI/SM/1 queue with ten customer types, with $k = 64$ for the Intel i860 and RIOS experiments, and $k = 32$ for the others. The performance results for this experiment are shown in Tables 6a and 6b. For the i860 and RIOS environments, $lt = 4000$, $ut = 50 \times 10^6$, $gsize = 500$, $\alpha = 0.05$, and $\gamma = 0.075$. For the others, $lt = 2000$, $ut = 50 \times 10^6$, $gsize = 100$, $\alpha = 0.05$, and $\gamma = 0.075$.

**Experiment 7**: *Simulating Unified Mutant Execution on SIMD Machines*

Program *unification* has recently been recognized as a viable technique for obtaining speedup across program execution [42]. Clearly, a program that vectorizes poorly cannot exploit the functional units of a vector machine (e.g., Alliant FX/80, Cray Y/MP). If such a program is required to be executed on several data sets (e.g., computing an integral

Hypercubes

| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|
| RX | 4492 | 4560 | 1499 | 659 | 308 | 151 | 76 | 38 |
| tail-size = 64 | 1.059m | 1.059m | 1.056m | 1.054m | 1.057m | 1.062m | 1.095m | 1.071m |
| IPSC | 3314 | 3337 | 1142 | 479 | 236 | 124 | 69 | NA |
| tail-size = 32 | 137.2k | 137.2k | 139.6k | 134.2k | 139.6k | 136.8k | 138.2k | NA |

Time/Sample-size for tail probability estimation (Intel Hypercubes)

**Table (6a)**

Other

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| SEQUENT | 6167 | 3096 | 2104 | 1563 | 1249 | 1052 | 894 |
| tail-size = 32 | 137.6k | 139.2k | 138.6k | 139.6k | 138.4k | 137.8k | 138.4k |
| RIOS LN | 15788 | 7899 | 5275 | 3956 | 3219 | 2834 | 2358 |
| tail-size = 64 | 1.055m | 1.059m | 1.056m | 1.054m | 1.057m | 1.062m | 1.095m |
| SUN4 LN | 3620 | 1842 | 1222 | 921 | 746 | 634 | 560 |
| tail-size = 32 | 136.2k | 137.4k | 137.2k | 138.4k | 136.4k | 139.2k | 140.2k |
| HWAN | 3626 | 1455 | 4462 | 3469 | 2479 | 2094 | 1575 |
| tail-size = 32 | 136.2k | 137.8k | 138.4k | 138.4k | 139.2k | 137.4k | 137.8k |

Time/Sample-size for tail probability estimation (Sequent, LNs, HWANs)

**Table (6b)**

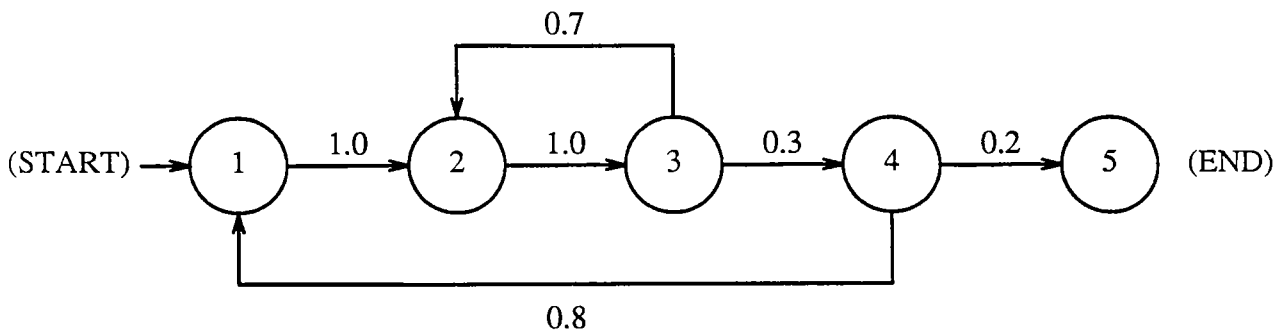


Figure 1: Flow graph of a five-node program

numerically for several values of a parameter), it is possible to create a single program which, when executed, will realize all possible executions of the original program on the different data sets. Such a *unified* program is obtained through an elementary transformation process [42] and can be shown to be highly vectorizable.

Consider the flow graph shown in Figure 1 representing a simple five-node program $P$. If each program block is viewed as an urn, and each arc (with associated probability) represents a possible execution path, then one can mentally simulate this program's execution by imagining a single ball, originally placed in Urn 1, moving through the flow graph randomly, until it finally reaches Urn 5 and terminates.

Unification may easily explained through the above urn model. Instead of working with a single ball, one thinks of $N$ balls initially in Urn 1 (where each ball represents a program working on its own data set). Due to vectorization, the amount of time spent by $m$ balls in any urn is less than $m$ times the amount of time spent by a single ball in any urn, for $1 < m \leq N$. The savings to be had by executing $m$ identical program blocks together increases with $m$, though with diminishing returns. Nevertheless, the savings are sufficient to yield speedup, and this is the key idea behind program unification.

The $N$ balls are made to move randomly through the program graph. The intention is to get all $N$ balls into urn 5 and to completion as fast as possible. Since different balls may traverse different paths, actively reflecting program paths determined by different data, it becomes necessary to choose a specific urn (for a vector uniprocessor) or set of urns (for

Hypercubes

| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|
| RX | 31216 | 31814 | 10410 | 4512 | 2394 | 1169 | 616 | 330 |
| | 30.15k | 30.15k | 30.09k | 30.26k | 30.35k | 30.60k | 30.75k | 30.90k |
| IPSC | 11142 | 11214 | 3842 | 1946 | 885 | 496 | 297 | NA |
| | 2690 | 2690 | 2660 | 2690 | 2730 | 2710 | 2820 | NA |

Time/Sample-size for SIMD Testing experiment (Intel Hypercubes)
**Table (7a)**

a vector multiprocessor) to move balls from at each step. This is precisely a scheduling problem. If we choose to throw those balls currently in some urn, say Urn $j$, then the vector uniprocessor being modelled currently executes those program components of the unified program which are residing in block $j$ of the original program.

The unification idea can be extended to enhance the execution of so called *program mutants* in software testing, as proposed in [32]. Mutation analysis [5] is a fault based testing technique which creates program mutants $P_1$ through $P_N$ of a given program $P$, via an application of mutant operators [30]; each of these mutants is syntactically different from $P$. In brief, if a set of test cases can distinguish these mutants from $P$, and if $P$ works correctly on these test cases, it is assumed that $P$ is reliable.

The $N$ mutants are programs almost identical to one another except for a small segment of (mutated) code in each which differentiates them from $P$. Certain computational problems arise in executing these mutants against given test cases. Due to the tendency of mutants to execute almost identical paths when fed identical test data, it was found that a simple variant of the unification idea described above performs favorably for achieving efficient mutant execution on SIMD machines. Details of the technique are beyond the scope of this paper and can be found in [31, 32].

In this experiment we simulate the execution of the mutant-unification algorithm on an SIMD machine. The operation of a fourteen block text-formatting program (see [32]) is simulated via the urn model, with number of balls ranging from 32 to 128, in steps of 32 (and hence each simulation experiment is made up of four sub-simulations). This experiment was considerably time-consuming, requiring more than eight and a half *hours* of cpu time on a single processor of the i860. In contrast, using all 128 processors of the i860, the entire simulation took only 5.5 *minutes*. The performance results for this experiment are shown in

Other

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| SEQUENT | 13828 | 7285 | 4284 | 3546 | 2796 | 2364 | 1985 |
| | 2690 | 2710 | 2730 | 2690 | 2720 | 2690 | 2720 |
| RIOS LN | 30496 | 15327 | 10365 | 7826 | 6120 | 5165 | 4367 |
| | 30.09k | 30.15k | 30.70k | 31.00k | 30.20k | 30.15k | 30.09k |
| SUN4 LN | 8118 | 4379 | 2887 | 2122 | 1705 | 1609 | 1364 |
| | 2690 | 2710 | 2660 | 2680 | 2720 | 2690 | 2690 |
| HWAN | 8240 | 3461 | 10145 | 8105 | 5596 | 5127 | 4451 |
| | 2690 | 2720 | 2660 | 2720 | 2690 | 2690 | 2700 |

Time/Sample-size for SIMD Testing experiment (Sequent, LNs, HWANs)
**Table (7b)**

Tables 7a and 7b. For the i860 and RIOS environments, parameters were fixed at $lt = 30$, $ut = 10^6$, $gsize = 10$, $\alpha = 0.01$, and $\gamma = 0.01$. For the others, $lt = 30$, $ut = 10^6$, $gsize = 10$, $\alpha = 0.04$, and $\gamma = 0.05$.

**Experiment 8:** *Simulating an FDDI Token Ring Network*

The Fiber Distributed Data Interface (FDDI) token ring [11] is, in essence, a token ring [6, 27] with a key additional parameter known as the *target token rotation time* (TTRT) which is a constant for the ring. Viewed as a queueing system, the FDDI ring is a multiqueue with a single cyclic server. While the token ring allows queues to have a service discipline that is independent of the token's cycle time [6], the FDDI ring requires that the token spend only a limited amount of time at a station. The FDDI ring has a built-in priority mechanism that handles two types of traffic, the high-priority *synchronous* traffic, and the low priority *asynchronous* traffic with up to eight priority levels in the latter. Through use of the TTRT, the FDDI medium access control protocol limits the number of data frames transmitted by a station on any cycle in order to guarantee service to synchronous traffic at stations.

The operation of the protocol is briefly described as follows. The station in possession of the unique token is allowed to transmit data frames for a certain amount of time which depends on network parameters. During its transmission, other stations merely forward

incoming frames downstream. The sending station finally removes its returning frames from the ring and passes the token on, immediately after its last frame has been sent, to the next station on the ring where the process just described is repeated if the station has frames to transmit. During ring initialization stations negotiate a satisfactory common TTRT value. The TTRT is defined to be the *maximum average cycle-time* of the token. It is known [24] that the maximum cycle-time of the token is bounded from above by twice the TTRT value; stations with stringent cycle-time requirements can negotiate accordingly. The TTRT decided upon finally is the smallest TTRT value requested by a negotiating station.

Each station utilizes a local timer, called a *token-rotation timer* (TRT), to measure the time between consecutive visits of the token. On each visit of the token, the TRT is reset so as to measure the next cycle-time of the token. Each time that a station determines that the token is late, which is the case if it finds TRT > TTRT, it computes the amount of time by which the token is late and adds this amount to its next TRT so as to accumulate the effects of late token arrivals.

On each visit of the token, a station may transmit an amount of synchronous traffic limited to some predetermined fraction of the bandwidth, where this fraction is obtained from the station management entity. Following this, each station utilizes another local timer, called a *token-holding timer* (THT) to control the amount of time allocated to the transmission of asynchronous frames. Each of the eight classes of asynchronous traffic is allocated a certain amount of transmission time. The THT, which is initially loaded with the value of (TTRT − TRT) when the token is early, is decremented appropriately as each priority class consumes its share of transmission time. When its THT expires, a station passes the token on to the next station on the ring. If the token arrives at a station late, the station merely transmits its synchronous traffic and relinquishes the token.

In this experiment we use regenerative methods to simulate an FDDI token ring with five-hundred nodes on the i860 and RIOS environments, and fifty nodes on the other environments. The simulation model measures message queueing delay at each station. Once again, parameters were set so that the resulting simulation was time-consuming. The performance results for this experiment are shown in Tables 8a and 8b. For the i860 and RIOS environments, parameters were fixed at $lt = 10$, $ut = 10^5$, $gsize = 10$, $\alpha = 0.01$, and $\gamma = 0.01$. For the others $lt = 10$, $ut = 10^5$, $gsize = 10$, $\alpha = 0.01$, and $\gamma = 0.025$.

Hypercubes

| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|
| RX | 35006 | 35210 | 12240 | 5346 | 2510 | 1280 | 630 | 332 |
| | 31.8k | 31.8k | 32.8k | 31.8k | 31.6k | 32.2k | 32.2k | 32.4k |
| IPSC | 32400 | 32410 | 10814 | 4620 | 2082 | 1066 | 586 | NA |
| | 28.5k | 28.5k | 28.2k | 26.6k | 27.2k | 28.2k | 28.0k | NA |

Time/Sample-size for FDDI Token Ring (Intel Hypercubes)

**Table (8a)**

Other

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| SEQUENT | 43416 | 22843 | 14566 | 10963 | 8684 | 7327 | 6200 |
| | 28.4k | 28.2k | 28.2k | 27.8k | 28.2k | 28.0k | 27.2k |
| RIOS LN | 34325 | 17240 | 12224 | 8590 | 6990 | 5810 | 5010 |
| | 32.4k | 31.8k | 31.8k | 31.8k | 32.2k | 31.8k | 32.4k |
| SUN4 LN | 36520 | 19100 | 12200 | 9310 | 7386 | 6112 | 5210 |
| | 28.4k | 28.4k | 28.2k | 28.2k | 26.5k | 27.4k | 27.2k |
| HWAN | 35640 | 19800 | 39680 | 36100 | 26590 | 21200 | 18450 |
| | 28.1k | 28.4k | 28.4k | 28.2k | 28.4k | 28.2k | 28.4k |

Time/Sample-size for FDDI Token Ring (Sequent, LNs, HWANs)

**Table (8b)**

# 6 Conclusion

Our initial experiences with the *EcliPSe* toolkit have been more than satisfactory. We find it amply rewarding to be able to take a simulation program written for a sequential machine and execute it, with the help of the toolkit, on a multiprocessor environment. The resulting execution time is usually a small fraction of the model's execution time on a uniprocessor. For example, the five-hundred station FDDI token ring simulation (Experiment 8) required roughly 9.7 *hours* on a single processor, and as little as 5.5 *minutes* using all 128 processors of an Intel i860 hypercube.

Let us conclude by saying that the data-parallel programming paradigm for concurrent simulation has demonstrated potential benefits and related problems that are worthy of detailed investigation. These investigations could involve schemes for improving speedup in situations where sampling times are highly variable (e.g., Hitting-times in Markov chains), methods for adding/deleting processors when necessary, improving fault-tolerance of long-running simulations, methods for constructing efficient estimators, techniques for efficient minimal functional decomposition of large models, etc. It is hoped that our research in this direction will stimulate further work, to make careful but compute-intensive simulating a routine part of an experimental scientist's repertoire.

**Acknowledgement**

The authors are indebted to Aditya P. Mathur, poet, composer, friend, and teacher, for providing the motivating ideas and encouragement for this work, and without whose most recently acquired penchant for typesetting, we may have forever been lost in the austere world of fonts. We are also grateful to Mike Heath for his continued support and encouragement.

## References


[1] Aarts, E., and Korst, J., *Simulated Annealing and Boltzmann Machines,* John Wiley and Sons, 1989.

[2] Bell, G., "The Future of High Performance Computers in Science and Engineering," *CACM,* Vol. 32, No. 9, pp. 1091-1101, September, 1989.

[3] Bhavsar, V., and Isaac, J., "Design and Analysis of Parallel Monte Carlo Algorithms," *SIAM J. Sci. Stat. Comput.,* Vol. 8, No. 1, pp. 73-95, 1987.

[4] Biles, W., Daniels, D., and O'Donnell, T., "Statistical considerations in simulation on a network of microcomputers," *Proceedings of the 1985 Winter Simulation Conference*, pp. 388-393, December 1985.

[5] Budd, T. A., *Mutation Analysis of Program Test Data,* Ph.D. Thesis, Yale University, New Haven, CT, 1980.

[6] Bux, W., "Local Area Subnetworks: A Performance Comparison," *IEEE Transactions on Communications*, Vol. 29, No, 10, pp. 1465-1473, October 1981.

[7] Crane, M., and Iglehart, D., "Simulating Stable Stochastic Systems, III: Regenerative processes and discrete event simulations," *Operations Research*, Vol. 23, pp. 33-45, 1975.

[8] Chamberlain, R., and Franklin, M., "Hierarchical discrete-event simulation on Hypercube Architectures," IEEE Micro, pp.10-20, August 1990.

[9] Cross, G., and Jain, A., "Markov random field texture models," *IEEE Trans. PAMI-5,* pp. 25-39, 1983.

[10] Dijkstra, E., W., "Self-stabilizing systems in spite of distributed control," *CACM,* Vol. 17, No.11, pp. 643-644, 1974.

[11] Dykeman, D., and Bux, W., "An investigation of the FDDI media-access control protocol," *Proceedings of the Fifth European Fibre Optic Communications and Local Area Networks Exposition,* June 3-5, 1987.

[12] Feynman, R., and Hibbs, R., *Quantum Mechanics and Path Integrals,* McGraw-Hill, N.Y., 1965.

[13] Fishman, G., *Concepts and Methods in Discrete Digital Simulation*, Wiley, New York, 1973.

[14] Fok, D. S., and Crevier, D., "Volume Estimation by Monte Carlo Methods," *Journal of Statistical Computation and Simulation*, Vol. 31, pp. 223-235, 1989.

[15] Fox, G., "Parallel computing comes of age: supercomputer level parallel computations at Caltech," *Concurrency: Practice and Experience,* Vol. 1, No. 1, pp. 63-103, 1989.

[16] Fox, G., and Messina, P., "Report for 1988 on the Caltech Concurrent Computation program", *Annual Report C3P-685,* California Inst. of Technology, Dec. 1988.

[17] Fujimoto, R., "Parallel Discrete Event Simulation," *CACM,* Vol. 33, No. 10, pp.30-53, October 1990.

[18] Gay, J. *et al,* "Component Placement in VLSI Circuits Using a Constant-Pressure Monte Carlo Method," *Integration, the VLSI Journal,* No. 3, pp. 271-282, 1985.

[19] Goldberg, D., *Genetic Algorithms,* Addison Wesley, 1989.

[20] Heath, M. T., Ng, E., and Peyton, B. W., "Parallel algorithms for sparse linear systems", *SIAM Review*, Vol. 33, to appear, 1991.

[21] Heidelberger, P., "Discrete Event Simulations and Parallel Processing: Statistical Properties," *SIAM J. Statist. Comput.*, Vol. 9, No. 6, pp. 1114-1132, 1988.

[22] Hofri, M., *Probabilistic Analysis of Algorithms* , Springer-Verlag, 1987.

[23] Householder, A., S., *Principles of Numerical Analysis*, McGraw-Hill, New York, 1953.

[24] Johnson, M. J., "Proof that Timing Requirements of the FDDI Token Ring Protocol are Satisfied," TR 85.8, RIACS, NASA Ames Research Center, August 1985.

[25] Kalos, M., and Whitlock, P., *Monte Carlo Methods,* Wiley, N. Y., 1986.

[26] Kirkpatrick, S., Gelatt, C., and Vecchi, M., "Optimization by Simulated Annealing," *Science,* No. 220, pp. 671-680, 1983.

[27] Kuehn, P. J., "Multiqueue Systems with Nonexhaustive Cyclic Service," *Bell Systems Technical Journal*, Vol. 58, No. 3, pp. 671-698, March 1979.

[28] Lubachevsky, B., "Efficient distributed event-driven simulations of multiple-loop networks," *CACM,* Vol. 32, pp. 111-123, January 1989.

[29] MacDougall, M., *Simulating Computer Systems,* MIT Press, 1987.

[30] Mathur, A. P., *et al* , "Design of Mutant Operators for the C Programming Language," SERC-TR-41-P, Purdue university, 1989.

[31] Mathur, A. P., Krauser, E., and Rego, V., "Mutant Unification: A New Method for Mutation Testing On SIMD machines," *Proceedings of the International Conference on Software Engineering*, Toulouse, France, December 3-7, 1990.

[32] Mathur, A. P., Krauser, E., and Rego, V., "High Performance Software Testing on SIMD Machines," *IEEE Transactions in Software Engineering*, to appear, 1991.

[33] Meier, D. *et al*, "A general framework for complex time-driven simulations on hypercubes," Caltech TR C3P-71, CIT, March 1989.

[34] Neuts, M. F., *Matrix-Geometric Solutions in Stochastic Models: An Algorithmic Approach*, The Johns Hopkins University Press, 1981.

[35] Neuts, M. F., "Some explicit formulas for the steady-state behavior of the queue with semi-Markovian service times," *Adv. Appl. Prob.*, Vol. 9, pp. 141-157, 1977.

[36] Pearl, J., *Probabilistic Reasoning In Intelligent Systems*, Morgan Kaufmann, 1988.

[37] Parekh, S. and Walrand, J., "A Quick Simulation Method for Excessive Backlogs in Networks of Queues," *IEEE Trans. on Automatic Control*, Vol. 34, No. 1, pp. 54-66, January 1989.

[38] Reed, D., *et al*, "Parallel Discrete Event Simulation: A Shared Memory Approach," *Proc. SIGMETRICS*, pp. 36-38, 1987.

[39] Reed, D., *et al*, "Parallel Discrete Event Simulation Using Shared Memory," *IEEE TSE*, Vol. 14, No. 4, pp. 541-553, 1988.

[40] Rego, V., "A Band and Bound Technique for Simple Random Algorithms," *Probability in the Engineering and Information Sciences*, Vol. 4, pp. 333-344, 1990.

[41] Rego, V., Chuang, L., and Mathur, A. P., "Concurent Stochastic Simulations: Experiments with Unification," *Proc. Fifth Canadian Supercomputing Symposium*, June 3-5, Fredericton, N.B., Canada, 1991.

[42] Rego, V., and Mathur, A. P., "Exploiting Parallelism Across Program Execution: A Unification Technique and Its Analysis," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 4, pp. 399-414, October 1990.

[43] Ripley, B., *Stochastic Simulation*, Wiley, N.Y., 1987.

[44] Rubinstein, R., *Simulation and the Monte Carlo Method*, Wiley, N.Y., 1981.

[45] Schmeiser, B., "Generation of the Maximum (Minimum) Value in Digital Computer Simulation," *J. Statist. Comput. Simul.,* Vol. 8, pp. 103-115, 1978.

[46] Sunderam, V., and Rego, V., "*EcliPSe*: A System for High Performance Concurrent Simulation," to appear in *Software Practice and Experience*, 1991.

[47] Trivedi, K. S., *Probability and Statistics with Reliability, Queueing, and Computer Science Applications,* Prentice-Hall, 1982.

[48] Wagner, D., and Lazowska, E., "Parallel Simulation of Queueing Networks: Limitations and Potentials," *Proceedings of ACM Sigmetrics/Performance '89*, pp. 146-155, May 1989.

[49] Wallquist, A. *et al* , "Exploiting Physical Parallelism Using Supercomputers: Two Examples from Chemical Physics," *IEEE Computer,* Vol. 2, No. 5, pp. 9-21, 1987.