1990

ECLIPSE: A System for High Performance Concurrent Simulation

V. S. Sunderam

Vernon J. Rego

Purdue University, rego@cs.purdue.edu

Report Number:

90-1027

Sunderam, V. S. and Rego, Vernon J., "ECLIPSE: A System for High Performance Concurrent Simulation" (1990). *Department of Computer Science Technical Reports*. Paper 29.
<https://docs.lib.purdue.edu/cstech/29>

ECLIPSE: A SYSTEM FOR HIGH
PERFORMANCE CONCURRENT SIMULATION

V. S. Sunderam

Vernon J. Rego

CSD-TR 1027

October 1990

(Revised April 1991)

EclIPSe: A System for High Performance Concurrent Simulation*

V. S. Sunderam

Department of Mathematics & Computer Science
Emory University, Atlanta, GA 30322, USA

Vernon J. Rego[†]

Department of Computer Sciences
Purdue University, West Lafayette, IN 47906

Abstract

A description of a system which provides for the concurrent execution of stochastic simulation applications is presented. The *EclIPSe* system provides high level simulation primitives that enable common simulation tasks. An application programmer describes the simulation process in terms of these primitives using a sequential computation model. The system transparently replicates appropriate sections of the simulation when the program is executed on a concurrent system. By binding to machine-dependent versions of the *EclIPSe* library, the simulation program can execute without modification on a variety of architectures including uniprocessors, hypercubes, shared-memory machines, and loosely coupled networks. In those experiments conducted, near-linear speedups were obtained. The main design aspects of the system, salient implementation features, and performance figures for some simulation applications are presented. The usefulness of the system for more general applications is also discussed.

Keywords: simulation, concurrency, estimator, stochastic, regenerative, speedup.

* Research 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

Stochastic simulation [1] is an important tool that assists in the understanding of various complex phenomena. The number and diversity of applications is growing; representative examples are molecular dynamics methods in chemical physics [2], lattice theory problems [3], Monte Carlo methods [4], genetic algorithms [5], and simulated annealing [6]. Simulation is computationally intensive and typical applications often execute for hours or even days on fast scalar supercomputers. From another viewpoint, recent experiences with simulation applications have stressed the importance of being able to interact with the simulation program, e.g. for visualization of the intermediate results, and for providing inputs or altering the course of a run. The need for significantly improving the execution time of simulation programs is therefore critical in a variety of disciplines. Consequently, techniques for the exploitation of parallel machines for the execution of simulation programs is of considerable interest.

In recent years, researchers have suggested various techniques for multiprocessor based stochastic simulation. An example of a methodology oriented approach is the well-known Bryant-Chandy-Misra message passing paradigm of *distributed simulation*. In this method ([7], [8], [9]), the simulation model is functionally decomposed into components, each of which is executed on a different processor. The processors are interconnected by a topology that is compatible with the logical underpinnings of the simulated system. While this approach delivers reasonable speedup, most simulation models are not easily partitionable in this manner, or can lead to undesirable effects when decomposed. For example, queueing network models that behave well on uniprocessors are prone to deadlock when executed on distributed memory multiprocessors, as reported in [9] and [10-11]. Further, the limits to speedup attainable with this approach are dependent on factors other than the model alone, as reported in [12,13]. Overall, both the theoretical and empirical results of distributed simulation using functional decomposition have not been very encouraging. Another approach, with somewhat more encouraging results, is *parallel simulation* [12,13], where the functionally decomposed components are executed on a shared memory multiprocessor. In particular, when the workload is decomposed in symmetric manner, this approach has been found to work reasonably well [14,15]. Yet another distributed approach is the *Time Warp* algorithm which operates by ignoring inter-processor synchronization issues, making compensations after the fact by performing rollbacks to a consistent state. This method has yielded good performance when rollbacks are infrequent [16,17] but suffers from

problems (e.g., error propagation, state-saving, debugging complexity) some of which are inherent to the distributed simulation approach.

The distributed and parallel simulation techniques described above have mainly dealt with discrete event simulation (in particular, queueing network models) on MIMD architectures. While functional decomposition is effective in some cases and under the right circumstances [18], this mode of concurrency can only complement data parallelism [3]. Motivated by our own experiences and that of other researchers who have felt the need for alternative multiprocessor simulation methodologies, the *Eclips_e* system has been developed to overcome some of the limitations of distributed and parallel simulation schemes. The *Eclips_e* system for concurrent[†] simulation is conceptually simple, yet powerful and flexible enough to accommodate a wide range of simulation applications and varied architectural bases. The *Eclips_e* prototype primarily supports the data-parallel paradigm, but contains provisions for simulation using functional decomposition as well. Simulation applications in this system may be implemented with much less effort than in distributed simulation schemes, are statistically efficient, and may execute on both shared and distributed-memory MIMD machines, uniprocessors, loosely coupled networks of workstations, and SIMD/vector computers.

The overall objective of this work was to create a set of concurrent simulation tools that enabled the rapid construction of efficient simulation systems. A primary goal was to ensure that simulation software built using these tools is portable across a variety of architectures and machine types. This was considered an important factor since typically, researchers have access to various types and classes of computing resources, and particularly for long-running, compute intensive simulation applications, frequently wish to execute their programs on different machines depending upon need and availability. Another important goal was to present an interface that is simple and straightforward, but sophisticated enough to allow the description of application specific nuances that do not conform to the "standard" simulation paradigm. In particular, it was desired that the user interface did not necessitate the specification of low level or

[†] The term concurrent is used in a more general sense than as defined in Jones [15]. Here, as in [15], simulation workload is decomposed symmetrically, all processes execute identical code, no process has a fixed association with a logical process being simulated, and the number of processes may be changed arbitrarily without affecting the system being simulated. However, since we felt that architectural issues must be separated from the chief goals of multiprocessor simulation, we allow the term concurrent to envelop arbitrary architectures.

machine dependent details, especially those relating to communication and synchronization. On the other hand, it is possible that there exist applications with very specific and extraordinary control structure and performance characteristics — designing a user interface that was flexible enough to cater to such needs was an important design objective. A third goal was to support as much fault tolerance as the application and the target machine would permit. Since simulations are extremely compute intensive and long-running applications, resilience to failures is an important and desirable feature. Process faults, processor failures, or communication failures on multiprocessors should not only permit the simulation to continue with degraded performance, but the possibility of restarting or relocating the failed component should also be explored where possible. Finally, recognizing that graphical interfaces — both for specification of the simulation system as well as for interacting with an executing simulation — are frequently desired, a project goal was to enable this to the maximum extent possible.

The *Eclipse* system consists of three major components. The first is a collection of application interface constructs, in terms of which a simulation system is described. The majority of these constructs fall into two main categories — one comprising commonly required simulation functions such as random number generation, statistics calculation, confidence interval testing etc. ; and the other consisting of primitives that permit the specification of the control structure of the simulation. This set of constructs is complete for a large class of stochastic simulations, but is nevertheless extensible, *i.e.*, the application programmer may add to this repertoire, or may directly code alternatives in the host programming language where appropriate. The constructs provided for control structure specification are organized in a manner that presents different levels of abstraction to the user. The *Level 0* constructs comprise the simplest set. A programmer using this level specifies the simulation system as if it were to execute on a sequential machine; this specification is converted transparently by the *Eclipse* system into a program in which appropriate sections are executed concurrently. At the next level of detail (*Level 1*), the programmer is aware of the existence of multiple *processes*, but is not concerned with the underlying architecture or the parallel computation model. This permits a user to perform explicit partitioning, data placement, and identification of certain instances of the concurrent processes for special tasks, without the need for concern regarding low level machine dependent details. Finally, the *Level 2* interfaces provide the facility for simulation specification with explicit knowledge of the target machine architecture. This level will be used by application programmers who wish to utilize only the simulation oriented routines (e.g. random number generators, confidence

interval testers) provided by *EclIPSe*.

The second component of the *EclIPSe* system is the translator — a tool that processes the simulation description (written in a high level language augmented by the constructs mentioned above) and generates architecture dependent variants of the program(s) depending upon the intended target machine. By binding many of the machine dependent aspects of the simulation during this phase, execution time efficiency is improved while retaining portability. The final component of the system is a set of run-time libraries that perform the required simulation and control functions during the execution of the simulation. The libraries also include tracing and profiling support, with the aid of which post-mortem performance analysis and identification of hot spots may be done. When the targeted execution environment is a loosely coupled network, the libraries also contain support for detecting network or processing element failures and restarting or relocating failed components of a simulation on an alternative processor.

While the approach to concurrent simulation that has been adopted is conceptually simple, it is effective and sufficiently powerful for many real simulation applications as our experiences have shown. The *EclIPSe* system has been designed, and a prototype version has been implemented on four different machine architectures. A variety of applications ranging from Monte Carlo optimization to FDDI network simulations have been developed and executed under the *EclIPSe* system, and the results compared against both sequential versions and manually developed multiprocessor versions. The detailed results of these experiments are reported in a companion paper [19]; this paper describes our approach from the system point of view. The programming interface is described in detail in the next section, following which the design and salient implementation aspects are discussed. Representative results from a few simulation systems are then reported, and the concluding section discusses some of the critical issues in such an approach, the implications for applications other than stochastic simulation, and ongoing and future work.

2. The *EclIPSe* Programming Interface

The design of the programming interface in the *EclIPSe* system was guided by three fundamental considerations. The first is the "typical" structure of practically all

stochastic simulations — essentially involving repetitive sampling, statistics collection, computation of estimates, and testing for completion (e.g. by using confidence intervals). In addition to these standard activities, simulation programs also perform data and control input, result reporting, and miscellaneous functions such as initialization of random number stream seeds. The second factor that influenced the design of the programming primitives is the strongly felt need for graphical specification of the simulation system as well as for visualization of the results. The design of such an interface, that interfaces with the program level constructs, is in progress; the constructs themselves were developed in a manner that permits straightforward interfacing to the graphics components. Finally, simplicity and effectiveness with regard to such factors as portability, ease of use, and convenient debugging capabilities were considered important goals. The *Eclipse* primitives, with examples illustrating their use, are described in this section. The C language interface is used in this description; however, a Fortran interface is also available.

2.1. Simulation Constructs and Concurrency Issues

Simulations are important, frequently used applications and are performed by diverse groups of researchers in scientific, engineering, and other fields. There exist many computer programming languages specifically for simulation and several of these are in widespread use. Well known simulation languages include SIMSCRIPT [20], GPSS [21], SIMULA [22], and SLAM [23]. The language SLAM, a successor to GASP IV [24], is a good example of one that provides for the event-scheduling approach; i.e., a system is described in terms of its characteristic events (each of which is manifested in a routine), and the simulation proceeds by executing events in a time progression. The basic approach used by these and similar languages is that of providing to the user a standard framework in which the particular simulation may be described by following a few simple norms. A variety of examples may be found in the literature (e.g., [25]); the expressive power of these languages to describe a system leads to significant benefits in the development of a simulation.

While the approach of providing special simulation constructs adopted by existing simulation languages is effective, the model is not entirely appropriate when it is desired to parallelize a simulation. A variety of issues ranging from the semantics of input and output, termination control, proper estimation, and efficiency arise when

multiple threads of control are introduced. A few of these issues can be resolved when the simulation executes on a shared memory multiprocessor. Experiences with simulation on multiprocessors are reported in [11], [13], and [26]. The research described in [12,13] involves the use of *Synapse*, a distributed simulation environment that has been implemented as an extension of the PRESTO object oriented parallel programming system [27]. The *Synapse* approach is to provide abstract types that represent logical processes and simplex communication channels, in terms of which a programmer specifies the system to be simulated. A run-time scheduler ensures correctness and efficient execution, and assists in synchronization and communication.

The *Eclipse* system takes the view that a greater degree of parallelism is possible, in an *almost completely transparent* framework. Essentially, the user is required to provide only a *sequential* description of the simulation. This sequential specification is written in terms of an *Eclipse* defined, structured framework using special constructs; it describes the main components of the simulation (e.g. random number generators, samplers, confidence interval routines) and their interdependencies. The system transparently converts this specification to a parallel program, thus hiding details regarding the existence of, and communication between, multiple processes. However, sufficient flexibility is built into the framework to permit varying degrees of user participation in the parallelization of the application. The *Eclipse* programming model is based upon the conviction that implicit parallelism, with portability and low overheads, may be achieved in an effective manner. Our experience has shown that such an objective could be achieved by appropriately addressing input and output, communication, synchronization, and statistical issues.

2.2. Stochastic Simulation Primitives

The *Eclipse* system is partly based on the premise that stochastic simulation programs inherently possess a standard structure comprised of actions that may be encapsulated in well defined modules. The *Level 0* constructs in the system are derived from this model of stochastic simulation that consists of the following elements:

- Data Input : Some classes of simulation applications require certain types of input data. An example is simulation of a Markov chain in order to determine the

limiting probability of residing in a given state; the transition probability matrix comprises the data input for this application. In the solution of a system $Ax=b$ of linear equations using Monte Carlo methods, the A matrix and the b vector are data inputs. Simulation descriptions must therefore support the capability for this input function.

- Control Input : In addition to the data input described above, simulations often require additional inputs that are intrinsically of a different nature. Such inputs are termed *control* inputs; traditional examples are the confidence interval desired, upper and lower threshold values for the number of samples, seeds for the random number generators, types of statistics desired. The *Eclipse* system, in addition, optionally permits the specification of other control inputs in order to enable more effective parallelization. One example is *grainsize*, an indication of the number of samples to be generated before computing a statistic. Another is *seedsep*, defined as the distance between any two seed streams, which ensures that the simulation does not use duplicate seeds.
- Sample Generation : This activity is the core of the simulation, and generates a simulation sample. The sample is, in general, a vector and the generation process itself is application dependent. However, the *Eclipse* system provides a suite of common sample generators, and, for those applications that provide their own sample generators, supplies a collection of auxiliary functions (e.g., random number generators) that are frequently required. It should be noted that simulation applications based on the "event scheduling" approach also operate by generating samples although the sample itself, the distribution from which it is generated, and its semantics are somewhat different from the common notion of a sample.
- Statistics Combination : The samples generated in a simulation are usually combined in an application dependent manner to form an estimate. Various statistical factors influence the combining algorithm which is, in several cases, application dependent. Once again, the *Eclipse* system provides a comprehensive set of conventional statistics combination constructs.
- Termination Detection : The simulation process consists of generating samples and computing estimates until some termination condition is satisfied. Traditional examples are termination based on confidence intervals, or on a predetermined

number of samples. In *EclIPSe*, provisions are made for both program controlled termination as well as interactive decisions to terminate a simulation.

- Result Reporting : In a typical simulation application, a variety of results may be obtained, all of which are functions along different dimensions of the generated samples. Facilities for reporting standard, application dependent or run-specific results are provided.

2.3. The *EclIPSe* Model

An application programmer describes the structure of a simulation using the *EclIPSe* constructs embedded in a procedural *host* language. Constructs are provided for the specification of certain options, based upon which the translator and run-time libraries decide on the manner in which sections of the simulation are to be parallelized. The prototype primarily supports "data parallel" simulation, although certain classes of function parallel simulation also fit into the provided framework. In the data parallel model, multiple processes are used to generate simulation samples. The fundamental strategy in the *EclIPSe* system is to replicate a complete simulation program on multiple processors, and, with the support of the translation process and the run-time libraries, achieve synchronization, inter process communication and selective execution of actions such as input and termination detection. In the "central monitor" model, all samples are collected by a single control process which combines the samples, and tests for termination (based either on confidence intervals or predefined limits). The alternative paradigm is a "distributed monitor" scheme where each process performs both sample generation as well as combining and termination testing. The latter approach is useful in two situations. The first is when fault tolerance is desired; the impact of failure of the central monitor may be attenuated by replicating the statistics gathering and estimate computation at several sites. The second situation is when the simulation produces different types of results. In this case, each instance of the simulation program may compute a different statistic or subset of statistics, while different monitors focus on combining different statistics or subsets of statistics reported by the executing instances, thereby achieving a greater degree of parallelism.

The *Level 0* primitives in *EclIPSe* support the central monitor and replicated distributed models. At this level, the approach adopted is for each instance of the simulation

(a) Central Monitor Paradigm

(b) Distributed Monitor, Level 0

Figure 1: ECLIPSE Operational Model

program to obtain all the data structures used by the simulation in their entirety, and no attempt is made to partition the data. Different instances execute different sequences of the simulation as determined by the control inputs, e.g. different seeds. The *Level 1* primitives permit the programmer to participate to a greater extent in the parallelization of the simulation and to explicitly decompose data sets, or to specify different instances of the monitor as responsible for the computation of different statistics. The operational model used by the system is illustrated in Figure 1. In the remainder of this section, the primitives provided in the *Eclipse* system are described, and examples of simulation specifications at different levels are presented.

2.4. Initialization and Option Specification

The *Eclipse* system requires the specification of certain execution parameters before the constructs describing the simulation itself are used. These options are used by the translator and the run-time library to determine execution parameters and to optimize performance. For example, array sizes are determined by the translator in cases where the host language does not support dynamic allocation. Depending on the target architecture, the translator also defines communication and synchronization variables, based on the specified parameters. At run time, multiple processes must be initiated and communication channels set up – since these actions must be performed prior to any others, option specifications must appear before other *Eclipse* constructs.

An application program specifies these options using the `setoptions` construct, whose arguments are as shown below

```
setoptions(<number of processes>, <maximum number of samples>,  
          <number of items per sample>, <grainsize>, <monitor type/number>)
```

The first argument to this construct allows the user to specify the number of concurrent instances of the simulation. The special value "AUTOPROCS" may also be used, in which case the system decides on the number of processes based upon the execution environment. The user is also required to specify an upper bound on the number of samples and the number of distinct components in each sample. The grainsize

parameter is an indication to the system of the frequency at which samples are to be combined and tested for checking whether the termination condition has been met. The last two arguments are a flag that determines whether the centralized or distributed monitor model will be used and an indication of the number of monitors that are required. As an example

```
setoptions(16,1000000,7,100,DISTRIBUTED,4)
```

specifies that 16 instances of the simulation will execute, a maximum of one million samples will be generated, each sample is a vector of 7 elements, samples are to be combined after 100 samples have been computed by and received from each instance, and that the distributed monitor model is being used. The last argument specifies that 4 different statistics are to be computed; the system assigns responsibility for collection, combining, and termination detection for each of these statistics to four of the sixteen processes. When fault tolerance is also desired, the *<monitortype>* argument may be given the value "DISTALL", indicating that all processes will perform the statistics collection and combining functions. In the example above, this would result in a four fold replication of each of the four monitor functions.

2.5. Simulation Input

The input primitives provided in *Eclipse*, with examples illustrating their use, are presented in this section. This set of primitives is available in all three specification levels. The input constructs are organized in pairs, with one component specifying the type and size of the data structure, and the other requesting an input action. For data inputs, the **setupdata** primitive is used to indicate the nature of the data structure that is to be used (at present restricted to arrays) using the arguments as shown below

```
setupdata(<variable name>, <number of dimensions>, [<dimension1>, dimension2, ...>], <datatype>)
```

The **inputdata** primitive requests an input action. In order to provide flexibility, the input may be supplied interactively, from a file, or using a user supplied function. The form of this primitive is

```
inputdata(<sourcetype>, [<source>], [<format string>], <variable>)
```

As an example, consider the situation where a Markov chain simulation requires a transition matrix as input, the matrix being constructed on the fly by an application routine. The *Eclipse* constructs to be used in such a situation would be

```
setupdata(transmat, 3, 100, 100, 50, double)
```

```
inputdata(FUNCTION, matgen(x, y, transmat), transmat)
```

The semantics of the **setupdata** and **inputdata** primitives are as follows. If the specification is at *Level 0*, all data items are made available in their entirety to all instances of the program, although the actual input is performed only once. At *Level 1*, the input is done once when the above constructs are executed, but the system does not automatically distribute the data to the multiple instances. The programmer must use other constructs to explicitly transfer those portions that are required by the concurrently executing instances. These constructs are not used by *Level 2* specifications; at this level, data input, partitioning, and distribution must be coded in architecture and language dependent terms for a particular target machine.

In the *Eclipse* system, data inputs are distinguished from "control" inputs. The latter type of input is specific to a particular instance among the concurrent replications of the simulation programs, and is the primary method by which activity partitioning is achieved. The typical use of this construct in simulation applications is to assign different random number seed streams to different instances of the program, although the same facility may be used for other types of instance dependent data as well. The constructs for specifying and performing input of control data are **setupctl** and **inputctl** respectively. The argument lists for these constructs are identical to that for **setupdata** and **inputdata**; their use is illustrated by the following example:

```
setupctl(dseed, 0, double)
```

```
inputctl(FILE, "seedsfile", "%lf", dseed)
```

In the above example, the first construct specifies **dseed** as being a scalar, double precision, control variable. The second construct invokes the actual input function; once

again, physical input is performed only at one location. In this case however, the data is not replicated at all instances; the *EclIPSe* system performs multiple physical inputs, and ensures that each instance of the replicated simulation program receives a different value. As with the data input constructs, the input may be from a terminal or the values may be derived from a user specified subroutine.

It is interesting to note that this simple method of partitioning activities is useful in applications other than simulation, and is being enhanced further in our investigation of the applicability of *EclIPSe* to deterministic problems. In essence, this mechanism very conveniently provides the ability for static prescheduling, i.e., to divide a problem of size N among p processors. The same facility can be also used to allocate work to multiple processors in a dynamic fashion during execution, thus enabling multiprocessing using the "bag-of-tasks" approach with benefits in load balancing and greater overall throughput.

2.6. Simulation Constructs

The simulation paradigm that is supported by the *EclIPSe* prototype assumes a control structure consisting of sample generation, sample combining, and checking for termination. The sample generation process is the main activity and is the most computationally intensive component in stochastic simulations. The system provides built in sample generators for some classes of applications such as Markov chains, queueing chains, etc. These or, alternatively, user provided simulation routines may be invoked by the use of the `simulate` construct that specifies the particular sample generator required and its arguments. For example, in the simulation of a G/G/1 queue where an estimate of a queue length tail probability (i.e., the probability that, in steady-state, the queue size exceeds an integer value *tail*) is desired, the following call might be used:

```
simulate(ssqueue(dseed,mean_arrvt,mean_servt,  
               totsamples,tail,sample,cur))
```

The above example shows typical arguments to the `ssqueue` function, such as the mean arrival time, the mean service time, and the tail value. The last argument (`cur`) is a vector in which the sample generator returns a sample. An invocation of the sample generator above would typically be followed by a call to the construct


```
putstat(cur,0)
```

that stores the sample vector for further processing. The `putstat` construct essentially aggregates samples from different instances of the simulation program to enable combining, but these samples are not communicated to the combining process until "grainsize" samples are accumulated. The combining process is either the central monitor (indicated by the 0 argument in the example above) or that instance which is responsible for processing this particular statistic. Consider a situation in which the distributed monitor model is used to estimate two unknowns, with samples of the first unknown playing a part in generating samples of the second unknown. If the first unknown is of independent interest, statistics concerning this unknown may be reported at times which do not coincide with reports involving the second unknown. For example, suppose that an optimization problem for a random function must be solved for a minimum (sample), where the latter is used subsequently as a parameter in a stochastic maximization problem. If samples of the minimum are generated at a rate significantly different from the generation rate of samples in the maximization model, and if the minimum is of independent interest, then the following sequence of invocations might be used:

```
simulate(stoch_min(arg1,arg2,arg3,cur))
```

```
putstat(cur,1)
```

```
simulate(stoch_max(arg1,arg2,cur))
```

```
putstat(cur,2)
```

In this way, different monitors are able to construct estimates in an independent manner. In each case, the samples generated by multiple instances of the simulation program must be combined, using application dependent functions ranging from simple cumulative means to more complex functions such as bootstrapping techniques. The `combine` construct is provided to accomplish this; once again, the name of the appropriate functions and its arguments are specified in an invocation of this construct. For example, an invocation of the *Eclipse* provided combine function `cumul` might be written as

```
combine(cumul(mean, variance, totsample),1)
```

This routine, or a user provided combine routine extracts samples from the different instances of the simulation program, using the **getstat** construct and combines these to produce a single statistic. The system only executes the **combine** construct on those instances of processes whose function includes monitoring. The second argument to the **combine** construct identifies the appropriate statistic (and implicitly, the appropriate monitor) that is involved. After samples have been combined, the resulting statistic is inspected to determine if termination conditions have been met. The **termcheck** construct enables this, once again by using either a built in *Eclipse* routine or a user supplied termination detection routine. Termination detection routines must return a boolean result value to enable the system to take appropriate action. The **termcheck** construct also uses a parameter to identify the statistic based upon which termination is to be detected. It should be noted that when multiple statistics are being computed, termination based on one statistic causes the computation of only *that* result to be stopped. Samples for other statistics continue to be generated. As an example, the following invocation checks for termination conditions based on the confidence interval method:

```
termcheck(confid(left, right, mean, variance, precision,  
alpha, termflag),1)
```

Typically, the above constructs would be enclosed within a loop that would terminate when the termination condition was satisfied. Following this, the results of the simulation may be reported, using either the **report** or **simprint** constructs. The first construct consists of a collection of predefined report formats that is supported by *Eclipse*, while the latter permits the output of results that do not correspond to standard reports. The system also requires that the **simterminate** construct be invoked, to enable the run time libraries to perform certain epilog and housekeeping functions.

The primary constructs for controlling program execution have been described in the foregoing discussion. As mentioned, these constructs are embedded in a host procedural programming language to describe a stochastic simulation program; the translator and run-time libraries allow such a program to be executed on various types of uni- and multiprocessors without modification. A code skeleton illustrating the structure of an application for the stochastic solution of systems of equations using the covering

path method [28] is shown in Figure 2. It should be noted that this example illustrates an application with a deliberately simple control structure; more complex structures including dynamic adjustment of simulation parameters and use of multiple sample generators are also permitted.

```
...
/* Declarations, application dependent initializations */
...
setoptions(AUTOPROCS,100000,5,10,CENTRALMON,0)
/* Specify max samples, sample length, grainsize, monitortype */
...
setupdata(a,2,1000,1000,float)
/* Specify name, size and type of matrix */
...
/* Other setupdata constructs for precision, thresholds, etc */
...
inputdata(FUNCTION,createmat(a,1000,matrixseed))
/* Matrix to be created by user-supplied function */
...
/* Other inputdata invocations for precision, thresholds etc */
...
setupctl(dseed,0,double)
inputctl(FROMFILE,"seeds.1","%lf",dseed)
/* Set up and input random number stream seeds, different for different instances */
...
while (term == 0) {
    simulate(getpath(dseed,1000,a,...,totsample,cur))
    putstat(cur)
    combine(vector_comb(totsample,cur))
    termcheck(vector_confid(totsample,1000,cur,...,precision,term))
}
if (term > 0)
    report(NORMAL1,totsample,left,cur[0],right,cur[1])
if (totsample > upperthreshold) {
    simprint("Abnormal termination\n")
    simprint("No. of samples = %d\n",totsample)
    simprint("Estimate = %lf, variance = %lf\n",cur[0],variance)
}
...
simterminate()
...
```

Figure 2: Example simulation program in *Eclipse*

2.7. Level 1 Primitives

The *Eclipse* user interface constructs described in the preceding section are those used at the highest level of abstraction. However, for some applications, the computation model supported at this level may be restrictive, and it is possible that certain application requirements are difficult to express in these terms. In order to accommodate such requirements, the system provides a less abstract level of user interface primitives at which the existence of multiple processes is explicit. The constructs provided at this level, however, encapsulate the low level, machine dependent details of process and communication management, and permit applications to be ported from one machine environment to another without modifications. The set of constructs is minimal, and only contains facilities for process initiation, termination and simple synchronization and data communication. A brief outline of each construct is given below; portions of their syntax, semantics, and implementation have been derived from similar constructs in PICL [29] and the PVM system [30].

initiate(*<object file>*, [*<location>*])

causes the specified object file to be executed at the specified location¹. This construct returns an "instance number" that may be used in other constructs.

terminate(*<instance number>*)

terminates the specified instance of the simulation program.

senddata(*<type>*, *<data pointer>*, *<number of items>*, *<destination>*)

The **senddata** construct permits instances to communicate data values to another instance identified by its instance number. The *<type>* argument is used for translation when the simulation is being executed in a heterogeneous network environment.

recvdata(*<type>*, *<data pointer>*, *<source pointer>*)

This construct receives data values, from a specific source if one is specified. It returns the number of items of data received and the identity of the originator of

¹ Although *Level 1* specifications are transparent w.r.t. architecture and location, we have found that users often desire to specify these, especially in LAN/H-WAN environments. This one concession was made in *Level 1* to avoid necessitating hand-coding the entire simulation in architecture dependent terms (i.e., using *Level 2*).

the data.

waitfor(<event>)

permits the invoking process to wait until a particular event has occurred. It is intended as a primitive means of synchronizing between processes.

occur(<event>)

signals the occurrence of an event, enabling waiting processes to be resumed.

The primary advantage in using these constructs is that the *Eclipse* system supports efficient implementations of these primitives for a variety of machine architectures. These implementations also contain tracing and profiling support that could be valuable in optimizing or debugging.

2.8. Built-In Functions

In addition to the input, simulation control, and output constructs described in the previous section, the *Eclipse* system provides a large number of built-in library functions that are frequently required in stochastic simulations. Included in this repertory are routines for random number generation, standard sample generators, various statistic combination routines, and different types of confidence interval functions. Rather than provide an exhaustive list, the main classes of supported functions with representative examples in each are listed in this section.

Sample Generators

Since a variety of problems have fairly well-understood algorithmic solutions, it is not difficult to provide constructs and underlying data structures that generate samples in the solution of such problems. Examples include Markov chain related estimates (e.g., hitting times [28]), specific Queueing Systems (e.g., PH/PH/1 queues [31]), tools for stochastic optimization (e.g., stochastic maximum finding [1]), linear system solvers (e.g., Monte Carlo based solutions [32]), etc.

Random Number Generators

Facilities for the generation of random numbers from either the classical distributions [25] or empirical distributions (through user-supplied functions) are provided. Examples are generators based on distributions including Uniform, Exponential, Weibull, Gamma, Erlang, Normal, Binomial, Geometric, Poisson, etc. (see [25]). It is possible to generate phase-type [31] random variables of high-order due to *EclIPSe*'s capacity for concurrency. There is considerable scope for further work in efficient random number generation for *EclIPSe* in multiprocessor environments.

Statistics Combination

The careful simulation analyst is well aware of the problems associated with statistical estimation and interpretation of results in uniprocessor simulations. In multiprocessor settings, estimation problems and consequently interpretation problems become even more complex (e.g., [18]). In general, *EclIPSe*'s combining routines are simple, leaving the user considerable flexibility in constructing estimates. However, the system also provides standard routines for combining user supplied estimates from the various replicating instances on the monitor(s). Examples of these include routines for combining an equal number of samples from each replicated instance of the simulation (fair sampling), a fixed total number of samples (random completion time), a random total number of samples (fixed completion time) (e.g., [18]), maximizers, minimizers, etc.

Termination Detection

The type of termination detection mechanism used is largely a function of the kind of simulation being conducted. For long-running models that exhibit large relaxation times, appropriate *tests* [25] may be made for determining if the concurrent simulation has reached *steady-state*. This is typical of functionally decomposed models where the focus is on increasing the rate at which each sample path is generated. If steady-state tests are successful for such models, instead of immediate termination with some simple statistic, it may be necessary to initiate a *batching* [25] routine for constructing samples which can be reported to the monitor(s) for combining. In *EclIPSe*, the monitor(s) generally rely on confidence intervals with specified precision for terminating the entire model. A small variation of the procedure described above is used for *regenerative* simulations [25], which can require long-running times on uniprocessors, but shown to be good candidates for multiprocessor simulations. Classical sequential tests may also

be applied for termination.

2.9. The Translator and Extensibility

The translator component of the *Eclipse* system is a preprocessor that parses the constructs in a simulation program and generates environment-dependent calls to the built-in libraries. One of the functions of the translator is to handle the variable number of arguments that most of the constructs permit; these are converted by packaging the arguments appropriately. While some languages/compilers permit variable arguments inherently, it was decided to incorporate this functionality in the translator for portability reasons. Another responsibility of the translator is to generate code dependent on the model of concurrency supported by the target environment. Communication and synchronization between the multiple instances of a simulation program may be via shared memory or message passing; constructs such as `inputctl` and `putstat` are translated into appropriate shared memory access or data transmission and reception calls.

While some of the translator functions may be delayed until link- or run-time, we have encountered several instances where early binding is more effective or more manageable. For example, input statements with varying argument lists require run-time parsing that can be avoided by source level translation. As another example, the translator produces source declarations for communication channels (e.g. sockets for LAN/HWAN environments) and expresses communication actions in terms of these variables. Communication channels may vary in number and type (e.g. depending upon the number of monitors), and are therefore cumbersome to set up at execution time. The translator also attempts to perform some optimization based on simple heuristic rules. For example, in the program shown in Figure 2, the simulation requires the generation of a (dense) order 1000 matrix, by invoking a user supplied function. Our experiences suggest that in a distributed memory environment, it is more efficient to replicate the matrix generation process in all instances, rather than to generate it at one instance and communicate it to the others.

The transformations that are made by the translator are performed in a straightforward manner, using standard templates that are themselves expressed in terms of *Level 1* constructs. In addition to being effective, this strategy permits convenient modification

or addition to the set of constructs provided. Applications may define their own constructs or make appropriate changes to suit their special needs. While at present this must be done manually, the facility is nevertheless useful and permits a tailored evolution of the *EclIPSe* constructs.

2.10. Application Development in *Level 2*

As mentioned earlier, the *EclIPSe* system permits application specification at varying levels of abstraction. *Level 2* is the least abstract; indeed, an application written at this level may not contain any constructs that deal with process management, communication, or synchronization. This "level" is provided essentially to permit applications to use random number generators, sample generators, confidence interval routines, and other simulation oriented functions provided by the system. Communication and control structures must be manually coded for a specific target architecture.

3. Design and Implementation

One of the primary design goals of the *EclIPSe* project was to enable applications to be executed without source modification on a variety of machine environments. At present, the system supports uniprocessors, loosely coupled (local or wide area) networks of processors, hypercubes, and shared-memory multiprocessors; an SIMD implementation is under consideration. While the eventual goal is to provide a graphical specification language which would then be compiled for various target environments, the current strategy is to implement the constructs as machine-dependent libraries. The significant features of the *EclIPSe* implementations for the different target environments are described in this section.

3.1. Operational Overview

The primary functions of the support library routines is to manage the transfer of data between the multiple executing instances. The transfer of input data values is

Figure 3: Statistics Pool & Combination

straightforward, and only requires packaging the values appropriately — with conversions to machine independent form being required when executing on a heterogeneous loosely coupled network. The collection of statistics and their combination is somewhat more involved owing to the fact that samples may be generated at different rates. The strategy used in the *Eclipse* system is for each monitor process to maintain a statistics pool, into which the generated samples are inserted as they are available. In the case of distributed memory environments, samples are buffered and transmitted to the monitor process, at a rate controlled by the grainsize. The system also attempts to minimize the overheads in transmitting samples by dynamically adjusting the grainsize under certain conditions.

The statistics pool is organized as a three dimensional array, with one plane corresponding to each instance of the simulation program. At present, the built-in combination routines in the prototype require an equal number of samples from each instance; deviating from this requirement without other forms of compensation is known to lead to statistically incorrect results [18]. However, even with this restriction, there exists some flexibility in the manner in which statistics may be combined. An example scenario is shown in Figure 3; the heavy boxes show the different portions of the statistics pool array that may be used in combining samples. The combine routines obtain the statistics to be combined by invoking the `getstatrange` construct; the system returns the dimensions of that *rectangular* subblock of the statistics pool that contains the largest number of samples.

3.2. Hypercube Implementation

The *Eclipse* system has been implemented on the Intel iPSC/2 and i860 hypercube computers, and a variety of stochastic simulation applications have been executed in this environment. In the hypercube implementation, the translator converts the control flow constructs into code sequences that send and receive messages between the multiple instances of the simulation program. In the implementation of the *Level 0* primitives, only the node processors of the hypercube are used, with node 0 assuming responsibility for input and output. In the central monitor model, the statistics pool is also allocated within the process executing on node 0, which periodically receives messages containing samples from the other nodes and inserts them into this array.

The implementation of the distributed monitor model is somewhat more involved. In this model, samples must be exchanged between all those that perform statistics combining and termination detection functions. From the information supplied in the **setoptions** construct, the translator sets up a global table indicating the nodes at which simulation results are being computed; the run time libraries use this table to determine the destination(s) for the generated samples. Sending samples to specific monitor nodes continues until termination conditions based on that statistic have been met. It should be noted that at present, all samples that are generated are delivered to all monitor nodes; the distributed monitor facility is intended for those applications that compute different statistics based on a common set of samples. The *EclIPSe* system uses simple heuristics to decide whether samples are delivered by sequential transmissions to each monitor, broadcast from one monitor, or via dimensional exchange among the monitor nodes.

In both the central and distributed monitor models, an important decision to be made is whether monitor processes themselves execute the sample generation routines. An affirmative decision increases the total sample generation rate, but could lead to slowdown in the statistics combination and termination detection functions. In the distributed monitor model, monitors do participate, but at a reduced level when there are fewer monitors than processes. In other words, when the number of monitors is smaller than the total number of executing instances, the monitor processes invoke their sample generators every other cycle. Refinements of this strategy to be more responsive to the rate of sample generation, and similar strategies for the central monitor model are under consideration.

The other aspects of the hypercube implementation of *EclIPSe* are fairly straightforward. It should be pointed out however, that some of the system and software features supported by the Intel environment may not be available on other hypercubes, and therefore may hinder the implementation of *EclIPSe* on such systems. In particular, the ability to perform node I/O, and the facility for loading different object modules on different nodes on a subcube are likely to be areas in which difficulties may arise.

3.3. Shared Memory Multiprocessors

The *EclIPSe* system has been implemented on the Sequent shared memory multiprocessor (both the Balance & the Symmetry). The same model as in the hypercube implementation was used, with one process assuming responsibility for the input and output actions. This initial process spawns subsequent instances, by using standard system-supported facilities; these instances synchronize with the original process by using pipes. The transfer of data between the multiple executing instances and the monitor processes is achieved using shared memory. Shared memory segments are set up for large data structures as well as for the statistics pool. The latter structure is organized as described earlier; no locking is required because each sample generation process writes into one plane of the statistics pool array.

Some environment dependent problems were encountered in this implementation. Since monitor processes need to be notified when samples have been inserted into the shared statistics pool, frequent synchronization is required. The most efficient way of synchronizing was found to be via pipes¹; however, system imposed limitations on the number of file descriptors caused this scheme to fail when the number of processors was increased. An alternative method using asynchronous signals to the monitor processes was also investigated — while this scheme did not encounter the scaling limitations, it was observed that frequent context switches to the signal handlers caused a noticeable degradation in performance.

3.4. Loosely Coupled Networks

The *EclIPSe* system has also been implemented on loosely coupled networks of uniprocessor machines, and has been tested on both local and wide-area networks. The base implementation uses the commonly supported remote execution facility [33] that uses stream connections to communicate between initiating and initiated processes. While this scheme is effective and works well in most cases, it has some inherent

¹ Sequent shared memory multiprocessors do provide for synchronization using shared variables, in ALM (atomic lock memory) and in shadow cache. However, access is via spin locks that are expensive for *EclIPSe* applications, particularly when small granules are used.

drawbacks. Among the disadvantages are usual limitations on the number of simultaneous connections (limiting scalability), and the overheads of initial setup. More importantly, supporting the distributed monitor model when using this method is cumbersome and fairly complicated. The system provides optional support software that implements a connectionless, reliable protocol to overcome these limitations. This protocol is implemented in special support processes that must execute on the participating hosts. This protocol permits addressing of the multiple simulation processes using instance numbers, thereby enabling the use of the distributed monitor model, as well as *Level 1* primitives in a convenient manner.

The basic structure of the loosely coupled implementation of *EclIPSe* is very similar to that on the hypercube. The major differences are with respect to the fault tolerance aspects and the fact that substantial variations in sample generation rates are most likely to be observed in this environment. The system attempts to address the first issue by monitoring the status of all executing instances, and initiating replacement processes when failures are detected. In order to avoid wasting work performed by processes that subsequently fail, samples and their corresponding seed values are recorded, and replacement processes begin execution with the latest stable seed value from a failed process. This recovery action is optionally available; care must be taken to ensure that such semantics are appropriate for a given application.

The imbalance in the rate of generation of samples is usually a direct outcome of the varying processor speeds and external loads on the network, although the inherent nature of the sample generation process itself may be a contributing factor. For example, in collecting an equal numbers of samples (i.e., fair sampling) from each process, to avoid order-statistic related bias, monitor processes must attempt various combination strategies to achieve the best possible results. Our experience however, has shown that as simple a scheme as shown in Figure 3 is often adequate, especially in instances where sample generation rates differ by an order of magnitude. If a monitor can determine, by examining its statistics pool, that the sample generation rate is skewed, it is free to examine various ways of constructing an estimator using the fair sampling rule. In this way it avoids a slowdown due to slow processors by eliminating their samples entirely, while still yielding consistent and unbiased estimators.

More than in any other environment, the rate at which samples are communicated to the monitor processes in a loosely coupled environment can affect the performance of

an application. The grainsize parameter therefore has to be carefully chosen such that communication overheads are minimized, while avoiding the generation of unnecessary samples. An option in the system enables an adaptive mechanism to be used; the built in libraries attempt to optimize performance by dynamically adjusting grainsize as the simulation proceeds.

4. Empirical Results

The *Eclipse* system has been successfully used to implement and execute a large number of stochastic simulation programs. Detailed descriptions of the experiments themselves, an analysis of program characteristics, and comparisons between *Eclipse* versions and manually coded multiprocessor versions are reported in [19]. In this section representative examples are presented, with empirical measurements and speedup data.

In all the experiments described below, the combining method used was to require an equal number of samples from each replicated instance. Because of this, achievable speedup is governed by the sampling rate of the slowest executing instance, or the instance whose intervals between sample reports are dominating (i.e., it is possible for an instance executing on a faster processor to be bogged down with longer sample paths). Novel combining methods can avoid this problem, to some extent, by avoiding such instances. In spite of the combining strategy used in our experiments, the results have been very encouraging.

In the following experiments, the measured speedup $S(n)$ obtained when *Eclipse* uses n processors is given by the ratio

$$S(n) = \frac{T(1)}{T(n)} \quad (1)$$

where $T(n)$ is used to denote the time taken by the experiment when n processors are used, $n \geq 1$. 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 reported speedups conservative.

When each of *EclIPSe*'s instances executes on a different processor, $S(n)$ 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 $S(2)=1$ 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.

Hitting Times in Markov Chains

Markov chains are stochastic processes that are known to be useful in a variety of modelling contexts, such as the analysis of algorithms [34], system reliability [35], queueing applications [31], Monte Carlo based optimization [28], etc. Such chains are obtained through model specifics and either lend themselves to explicit results (formulae) or computational results (algorithms) which are intended to yield some insight into the behavior of the phenomena being modelled. In those instances where these approaches fail, such as when the transition probability matrix describing the chain is of high order (for example, the stochastic solution of large systems of equations), or instances where explicit or computational techniques are unavailable, the simulation of a chain becomes a viable alternative. This alternative becomes even more attractive in the context of multiprocessor simulation.

In the following experiment we generate sample paths of a large chain whose initial state is fixed, in order to estimate the hitting time to a fixed target state. The simulation terminates when a 99.9% confidence interval with a precision of 0.05 has been obtained by the monitor. As an example of the utility of this estimate, its reciprocal is an estimator for the steady-state probability of being in a certain state which takes on the role of both the initial state and the target state. These sample paths, also known as *passage times*, are typically highly variable and are responsible for a large variation in their generation times. The synchronization degradation so caused between sampling processors is directly related to the resulting speedup performance degradation. Despite this problem inherent feature, we report positive speedup results in this experiment.

This simulation application was implemented on the *EclIPSe* system and executed on hypercubes, shared memory machines, and networks of uniprocessors. On the Intel i860 hypercube and a local network of RIOS workstations, a 1000x1000 transition

probability matrix was used; a 500x500 matrix was employed for the other architectures. Table 1 shows the times required for this simulation, along with the speedups obtained, as a function of the number of processors. The entry titled "Misc. WAN" refers to a heterogeneous network of Sun, IBM Rios, and Sequent machines, located at three geographically dispersed sites. Since the processing capabilities of the computers used in the "Misc. WAN" experiment are very different, speedup ratios are not meaningful and thus are not given. The timings obtained in the Misc. WAN experiment, while apparently haphazard, are due to the varying speeds of the processors. For example, in going from one to two processors (see Table 1), the execution time is almost halved, but when a third (in our experiments the slowest) processor is added, the execution time exceeds that required for one processor. This is a consequence of the statistic combining rule used, where the monitor is required to obtain an equal number of samples from each processor (see also Figure 6). This suggests that, under such a rule, it is helpful to work with sampling processors of equal speeds. Alternatively, speedups can be improved by resorting to other estimation techniques [18].

As the figures in the table show, the speedups obtained in this experiment are modest to good. As explained above, each concurrently executing instance of the simulation exhibits a large variation in the amount of time it requires to generate a hitting time sample. If equal numbers of samples are required from each instance, a single late sample can adversely affect the overall running time of the application. It was observed during the experiments that some of the executing instances routinely reported samples at a slower rate than others, a phenomenon attributable to the lengths of the sample paths themselves and not on the execution speeds of the processors computing these samples. Particularly in heterogeneous environments, it was observed that (slower executing) sample generators which were unfortunate enough to take these paths often generated samples at less than half the rate of the others, on the average.

The apparent discrepancy between the speedup results on the two hypercubes deserves some clarification. The problem size on the iPSC/2 is smaller than on the iPSC/860 (i.e., 500 states versus 1000 states), and the `grainsize` parameter is larger on the iPSC/2 than on the iPSC/860 (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 iPSC/860 in comparison to the iPSC/2. Using `grainsize = 1` on the iPSC/860 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 ratio of $S(n)$ to n for the iPSC/860 to drop considerably.

No. of procs	1	2	4	8	16	32	64	128
iPSC/2 (Speedup)	270 1	277 0.97	93 2.90	41 6.58	22 12.27	14 19.28	6 45.0	NA NA
i860 (Speedup)	2407 1	2486 0.968	1001 2.40	553 4.35	290 8.3	150 16.40	91 26.45	57 42.22

(a) Intel iPSC and i860 hypercubes

No. of procs	1	2	3	4	5	6	7	8
Seq. Symmetry (Speedup)	239 1	132 1.81	95 2.51	69 3.46	55 4.34	45 5.31	39 6.12	34 7.02
RIOS Local Network (Speedup)	5910 1	3392 1.74	2315 2.55	1715 3.44	1388 4.25	1371 4.31	1146 5.15	1061 5.57
Misc. WAN	56	32	78	55	48	36	29	25

(b) Sequent Symmetry, IBM Rios workstations, Sun4+Rios+Sequent(uniprocessor)

Table 1: Times (in seconds) for Mean Hitting Time Simulation

Multi-dimensional Integrals/Integral Equations

An early and fruitful application of stochastic simulation, in the guise of the so called Monte Carlo class of methods, was in the estimation of integrals and the solution of integral equations [32]. The latter class of problems (e.g., the Dirichlet problem

[36]) involve the generation of sample paths in a Markov chain, and in this sense are similar to and involve techniques used in the previous experiment. In contrast, the estimation of multi-dimensional integrals can be done by using uniform random variates in a simple manner which does not exhibit the highly variable sampling time characteristics of the previous example.

In this experiment, we use *Eclipse* to estimate the value of the two-dimensional integral

$$E[I] = \int_0^{\infty} \int_0^x e^{-(x+y)} dy dx \quad (2)$$

obtaining an interval with a 99.99 % level of confidence and relative precision of 0.001. The estimate of the integral is viewed as the expectation of a random variable I , instances of which are generated via the sample-mean method. It should be noted that the dimensionality of the problem affects timing results only through the amount of time required to generate a sample. In other words, higher-dimensional integrals would require larger sample generation times. However, the sample generation times themselves would not exhibit variability across simulating processors, and consequently would not affect speedup results. In this sense, speedup results for the two-dimensional integral are reflective of speedup results in higher dimensions as well.

The results from this experiment are shown in Table 2. It can be seen that this application exhibits superior performance and speedup characteristics, owing to the low variability in the time required for the generation of each sample. Particularly for the Sequent shared memory multiprocessor, the speedup attained is nearly perfectly linear, since overheads incurred in communication between the replicated instances are negligible. As in the previous experiment, the results on the hypercubes deserve special attention. In this case, the problem size and the grainsize parameter (5000 samples per combination) are the same for both the iPSC/860 and the iPSC/2, but the random number seeds used on each machine were different. This caused the iPSC/860 to generate a few more samples than the iPSC/2 for some values of n . The results appear to suggest that, since each i860 processor is several times faster (roughly by a factor of 17) than an iPSC/2 processor, the granularity of computation is largely responsible for the relatively small drop in the $S(n)$ to n ratio, as n increases. However, this phenomenon requires further attention and is part of a more detailed study we plan to conduct.

No. of procs	1	2	4	8	16	32	64	128
iPSC/2 (Speedup)	12521 1	12726 0.98	4252 2.94	1811 6.91	851 14.71	411 30.46	202 61.98	NA
i860 (Speedup)	745 1	770 .96	275 2.70	118 6.31	56 13.30	27 27.59	13 57.30	7 106.42

(a) Intel iPSC and i860 hypercubes

No. of procs	1	2	3	4	5	6	7	8
Seq. Symmetry (Speedup)	6419 1	3213 1.99	2141 2.99	1615 3.97	1286 4.99	1073 5.98	921 6.96	823 7.79
RIOS Local Network (Speedup)	531 1	266 1.99	180 2.95	138 3.84	107 4.96	91 5.83	79 6.72	68 7.80
Misc. WAN	1498	620	2017	1496	1149	962	780	602

(b) Sequent Symmetry, IBM Rios workstations, Sun4+Rios+Sequent(uniprocessor)

Table 2: Times (in seconds) for Multi-dimensional Integral Estimation

Dijkstra's Self-Stabilization Algorithm

In computer science, the probabilistic analysis of algorithms has proven itself to be a field fraught with problems of considerable difficulty, especially in distributed settings. Simulation is a natural and powerful tool that can be used by an analyst to understand how distributed systems behave. A good example of an algorithm in a distributed setting is the self-stabilization algorithm proposed by Dijkstra [38]. When loosely coupled processors cooperate by exchanging messages with one another, processors are capable of taking the system into an erroneous state. One of a number of algorithms proposed by Dijkstra is the K -state algorithm, which takes a system of processors from an erroneous state back into an error-free state.

Dijkstra's algorithm can very briefly be outlined as follows. Assume that N processors are arranged in a unidirectional ring, with processors capable of receiving from their clockwise neighbours. Initially, each processor possesses an arbitrary integer label from the set $\{1, 2, \dots, K\}$, where $K > N$. Given a specific boolean function $B(L(i), L(j))$ where $j = i \bmod N + 1$, and $L(i)$ defines the label of processor i , we say that processor i is in "*trouble*" if the function evaluates to true. The system is said to be in an erroneous state if one or more processors is in *trouble*. The K -state algorithm allows each *troubled* processor i to asynchronously obtain label information $L(j)$ from its clockwise neighbour j with the intention of forcing $B(L(i), L(j))$ to become false, thereby ridding itself of its *trouble*. In so doing, it may cause its downstream neighbour to acquire a *troubled* status. By progressing through a sequence of such label acquiring actions by processors, the K -state algorithm ultimately brings the system into an error-free state. What is of interest to the analyst is the average execution time of the algorithm, as a function of N and K .

In the simulation exercise, we execute Dijkstra's algorithm on a ring of $N = 1000$ processors, with $K = 6000$. Each run of the algorithm comprises two separate phases. A setup phase is initiated in which processors are assigned labels in such a way that initially 100 processors out of N are in a *troubled* state. Next, the execution phase that actually simulates the K -state algorithm is initiated. The entire run yields one sample path of the stochastic process that we are interested in studying. The runs are repeated a large number of times so as to obtain a 95% confidence interval with a precision of 0.01. Table 3 shows the performance figures obtained by simulating Dijkstra's algorithm using the *Eclipse* system.

Once again, the speedups obtained are very good, with hardware multiprocessors exhibiting almost perfectly linear speedup. In this case, matching problem size and granularity are responsible for pleasing speedups that agree on both the hypercube architectures. The apparent small degree of superlinearity in the hypercube experiments is due to the coarseness of the timing. On the RIOS workstation network, this application performs reasonably well; monitoring showed that sub-linear speedup was due in most part to external load on the workstations, and only minimally due to communications overheads. It is instructive to note that some loss in speedup is due to the setup phase which is repeated for each sample path. If this phase is initiated for only the first sample path and kept fixed for the remainder, speedup results will improve due to a reduction in setup time.

No. of procs	1	2	4	8	16	32	64	128
iPSC/2 (Speedup)	9830 1	9835 0.99	3286 2.99	1477 6.65	646 15.21	321 30.62	157 62.61	NA
i860 (Speedup)	635 1	639 0.99	216 2.93	99 6.41	42 15.11	21 30.23	10 63.5	5 127

(a) Intel iPSC and i860 hypercubes

No. of procs	1	2	3	4	5	6	7	8
Seq. Symmetry (Speedup)	18395 1	9215 1.99	6164 2.98	4622 3.97	3773 4.87	3086 5.96	2633 6.98	2326 7.90
RIOS Local Network (Speedup)	3853 1	1967 1.95	1297 2.97	998 3.86	807 4.77	763 5.04	703 5.48	655 5.88

(b) Sequent Symmetry and IBM Rios workstations

Table 3: Times (in seconds) for Dijkstra's Self-Stabilization Algorithm

These experiments demonstrate the extremely attractive speedups attainable by simple strategies for concurrent stochastic simulation. Even for the hitting-time simulation, an application that exhibits significant load imbalance, efficiencies of 80% and above could be achieved. However, we do recognize that these results are not surprising, considering that many data parallel stochastic simulations belong to the class of "embarrassingly parallel" programs[3]. Nevertheless, this fact does not detract from the utility and value of the *Eclipse* system. The ability to develop simulation applications within a simple and straightforward framework, and execute them without any modification on a variety of environments is very valuable, particularly considering that little efficiency (if at all) is sacrificed in enabling this versatility.

5. Discussion

The *EclIPSe* system for high performance concurrent simulation has been described in the preceding sections. This system enables stochastic simulations — a category of applications with very intensive computation characteristics — to exploit concurrency in a straightforward manner. Although still in its infancy, the evolving *EclIPSe* system has reached a level of maturity that allows it to host a wide variety of simulation applications, only a few of which have been discussed above. Our experiences thus far have been very encouraging, and it is believed that the *EclIPSe* system will become a powerful tool to assist in an important class of computing applications.

5.1 Interaction and Graphics

One of the emerging issues in stochastic simulation is the importance of graphical aids in the modeling process, for model understanding [39], validation [40], and monitoring execution performance [41]. With this in mind, the *EclIPSe* system incorporates features that permit both dynamic and post-mortem visualization of various aspects of the simulation, and permits interaction with the simulation while it is in progress. Some of these features are currently under development, and analysis of many classes of stochastic simulation applications to determine appropriate visualization paradigms is in progress. In this section, a few examples of existing graphical interfaces are briefly described.

The statistics pool array is the principal source of data for graphical displays. As mentioned, the samples maintained in this array can be interpreted in a variety of ways to produce different information displays. For example, the convergence of an estimate or the concentrations of outliers may be displayed dynamically to provide information regarding the progress of the simulation. Figures 4 and 5 depict the manner in which the estimate and its variance in the Markov chain hitting time simulation converge to stable values. These displays were obtained by using *EclIPSe* provided interfaces to standard graph plotting utilities.

In addition to graphical displays that provide information inherent to the application, data regarding the status of execution and performance of the simulation program itself is often valuable. Particularly in a concurrent computing environment, this helps

Figure 4. Hitting Time Estimate

Figure 5. Hitting Time Variance

Figure 6. Various processor sampling rates

to identify bottlenecks and points of inefficiency, and may be used to tune the application. Figure 6 is an example of such a display. Once again, the application being monitored is the Markov chain simulation; the graph shows the rate at which samples are being generated by individual processors, while executing on a loosely coupled network of workstations. It can be seen in this graph that 2 of the 8 processors are generating samples at a considerably slower rate than the others. This has the undesirable effect of slowing down the overall simulation. Further, owing to flow control mechanisms in the communications protocol, the two "slow" processors periodically cause the other processors to suspend execution, as shown in the "staircase" pattern. A simulation user, monitoring this behavior, may decide to either suspend the two slow processors, or relocate those sample generators to other machines. Either action may be effected using *EclIPSe* provided utilities.

At a different level, the *EclIPSe* system provides for post-mortem analysis of various aspects of concurrent execution such as processor utilization, message delays, and communication overheads. This visualization is provided through the ParaGraph tool [41] based on trace information logged by the *EclIPSe* run-time libraries.

5.2 Non-Simulation Applications

While the *EclIPSe* system is primarily intended for use with stochastic simulation applications, some deterministic problems may also be solved within the same framework. Most data parallel applications fall into this category, since the system is general enough to accommodate any SPMD (single program multiple data) computation. However, there are two issues that must be considered while executing such deterministic computations under *EclIPSe*. The first is the expense of making data available at all processing locations. If this is done by replication, it is likely that the communication overheads would be prohibitive. For example, a naive implementation of matrix multiplication would involve the physical transfer of the matrices to be multiplied to each processing element, an action that could be extremely expensive. It should be noted that *EclIPSe* does provide an alternative; if matrix multiplication based on the pipe-multiply-roll algorithm [42] were used, the `control` input mechanism could be used to locate specific matrix bands on the appropriate processing elements. But this requires careful and explicit partitioning by the user and further, constraints on the number of processing elements used — a parameter that can normally be left unspecified in *EclIPSe*

programs.

The other issue in the use of *Eclips_e* for deterministic computations is the interaction between program instances. *Eclips_e* presently provides for two standard communication structures — the central and distributed monitors. Deterministic applications whose communication patterns do not fit well into either of these paradigms would be required to deviate from the *Eclips_e* structure or include other cumbersome mechanisms to operate. In order to address these issues and make the system more suitable for some classes of non-simulation applications, we are presently studying mechanisms that permit flexibility and versatility without violating the transparency aspects of the *Eclips_e* system. Automatic data partitioning, communications efficiency, irregular computation structures and processing element topologies are the main areas in which our current efforts are focused.

6. Conclusions

The *Eclips_e* system grew out of the need to increase the rate of execution of stochastic simulation applications, given that the hardware environments might change from one run to another. Rather than adopt a language based approach, the toolkit strategy described in this paper was adopted. This strategy led to substantially reduced development times, but more importantly, has permitted straightforward extensibility both in terms of supported constructs as well as in interfacing to other tools such as the graphical interface. The *Eclips_e* system has demonstrated that both flexibility in operating environments and high performance can be achieved for stochastic simulation applications by fairly straightforward exploitation of data parallelism. Further work on the *Eclips_e* system will concentrate on graphical specification methods, enhancements to the visualization facilities and partitioning aspects from the system point of view; and on statistical issues such as initialization bias, parallel random number generation, and asymmetric sample generation rates.

Acknowledgement

The authors are grateful to Mike Heath for encouraging them to pursue this work. The authors also thank the referees for a careful reading of the manuscript. Their comments greatly helped improve the presentation.

References

- [1] B. Ripley, *Stochastic Simulation*, Wiley, New York, 1987.
- [2] A. Wallquist *et al*, "Exploiting Physical Parallelism Using Supercomputers: Two Examples from Chemical Physics," *IEEE Computer*, Vol. 2, No. 5, pp. 9-21, 1987.
- [3] G. Fox, "Parallel computing comes of age: supercomputer level parallel computations at Caltech," *Concurrency: Practice and Experience*, Vol. 1, No. 1, pp. 63-103, 1989.
- [4] V. Bhavsar and J. Isaac, "Design and Analysis of Parallel Monte Carlo Algorithms, *SIAM J. Sci. Stat. Comput.*, Vol. 8, No. 1, pp. 73-95, 19
- [5] D. Goldberg, *Genetic Algorithms*, Addison Wesley, 1989.
- [6] S. Kirkpatrick, "Optimization by Simulated Annealing," *Science*, No. 220, pp. 671-680, 1983.
- [7] K. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *CACM*, Vol. 24, No. 3, pp. 198-206, 1981.
- [8] J. Misra, "Distributed Discrete-event Simulation," *Computing Surveys*, Vol. 18, No. 1, pp. 39-65, 1986
- [9] J. Peacock *et al*, "Distributed Simulation Using a Network of Processors," *Computer Networks*, Vol. 3, No. 1, pp. 44-56, 1979.
- [10] D. Reed, "Parallel Discrete Event Simulation: A Case Study," *Eighteenth Annual Simulation Symp.*, pp. 95-107, 1985.
- [11] D. Reed *et al*, "Parallel Discrete Event Simulation Using Shared Memory," *IEEE TSE*, Vol. 14, No. 4, pp. 541-553, 1988.
- [12] D. Wagner *et al*, "Techniques for Efficient Shared Memory Parallel Simulation," *Proceedings of 1989 SCS Multiconference on Distributed Simulation*, Simulation Series 21(2), pp. 29-37, March 1989.
- [13] D. Wagner and E. Lazowska, "Parallel Simulation of Queueing Networks: Limitations and Potentials," *Proc. of ACM SIGMETRICS/Performance '89*, pp. 146-155, May 1989.
- [14] S. Bruell *et al*, "Concurrent Simulation of Queueing Networks: Limitations and Potentials," University of Iowa, *CSTR preprint*, 1989.
- [15] D. Jones, "Concurrent Simulation: An Alternative to Distributed Simulation," *Proc. of Winter Simulation Conference*, pp. 417-423, 1986.
- [16] D. Jefferson, "Virtual Time," *ACM TOPLAS*, Vol. 7, No. 3., pp. 404-425, 1985.
- [17] D. Jefferson, *et al*, "Distributed Simulation and the Time Warp Operating System," *ACM OS Review*, 1987.
- [18] P. Heidelberger, "Statistical analysis of parallel simulations," *Proc. Winter Simulation Conference*, pp. 290-295, 1986.
- [19] V. J. Rego and V. S. Sunderam, "Experiments in Concurrent Stochastic Simulation: The *Eclipse* Paradigm," Purdue CSD-TR 91005, to appear in *Journal of Parallel and Distributed Computing*, 1991.
- [20] P. J. Kiviat *et al*, *The SIMSCRIPT II Programming Language*, Prentice-Hall, 1969.
- [21] G. Gordon, *The Application of GPSS V to Discrete Systems Simulation*, Prentice-Hall, 1975.
- [22] P. R. Hills, *An Introduction to Simulation using SIMULA*, Publication No. S55, Norwegian Computing Center, Oslo, 1973.

- [23] A. Pritsker and C. Pegden, *Introduction to Simulation and SLAM*, Systems Publishing, West Lafayette, IN, 1979.
- [24] A. Pritsker, *The GASP IV Simulation Language*, Wiley, 1974.
- [25] A. Law and D. Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, 1982.
- [26] R. M. Fujimoto, "Performance Measurements of Distributed Simulation Strategies," *Proc. 1988 SCS Multiconference on Distributed Simulation*, pp. 14-20, February 1988.
- [27] B. Bershad *et al*, "PRESTO: A System for Object-Oriented Parallel Programming," *Software: Practice and Experience*, Vol. 18, No. 8, pp. 713-732, 1988.
- [28] R. Y. Rubinstein, *Simulation and the Monte Carlo Method*, John Wiley & Sons, 1981.
- [29] G. Geist *et al*, "PICL: A Portable Instrumented Communication Library," ORNL/TM-11130, 1990.
- [30] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice & Experience*, Vol. 2, No. 4, pp. 315-339, December 1990.
- [31] M. F. Neuts, *Matrix-Geometric Solutions in Stochastic Models: An Algorithmic Approach*, The Johns Hopkins University Press, 1981.
- [32] J. M. Hammersley and D. C. Handscomb, *Monte Carlo Methods*, John Wiley & Sons, 1965.
- [33] "SunOS Programmers Manual," Sun Microsystems, 1988.
- [34] V. J. Rego, "A Band and Bound Technique for Simple Random Algorithms," *Probability in the Engineering and Informational Sciences*, Vol. 4, pp. 333-344, 1990.
- [35] K. S. Trivedi, *Probability & Statistics with Reliability, Queueing, and Computer Science Applications*, Prentice-Hall, 1982.
- [36] W. Kaplan, *Advanced Calculus*, Addison Wesley, Reading, MA, 1952.
- [37] S. E. Forsythe and R. A. Leibler, "Matrix Inversion by a Monte Carlo Method," *Math. Tables Other Aids Comput.*, Vol. 4, pp. 127-129, 1950.
- [38] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *CACM*, Vol. 17, No. 11, pp. 643-644, 1974.
- [39] B. Melamed and R. Morris, "Visual Simulation: The Performance Analysis Workstation," *IEEE Computer*, Vol. 8, pp. 87-94, 1985.
- [40] P. Gipps, "The Role of Computer Graphics in Validating Simulation Models," *Math. and Comp. in Simulation*, No. 28, pp. 285-289, 1986.
- [41] M. T. Heath and J. A. Etheridge, "Visualization of Parallel Algorithms," ORNL Technical Report, submitted to IEEE Software, February 1991.
- [42] G. C. Fox and S. W. Otto, "Matrix Algorithms on a Hypercube I: Matrix Multiplication," *Parallel Computing*, Vol. 4, pp. 17-31, 1987.