Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1995

Concurrent and Fail-Safe Replicated Simulations on Heterogeneous Networks: An Introduction to EcliPSE

Felipe Knop

Edward Mascarenhas

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

V. S. Sunderam

Report Number:

95-014

Knop, Felipe; Mascarenhas, Edward; Rego, Vernon J.; and Sunderam, V. S., "Concurrent and Fail-Safe Replicated Simulations on Heterogeneous Networks: An Introduction to EcliPSE" (1995). *Department of Computer Science Technical Reports.* Paper 1192.
https://docs.lib.purdue.edu/cstech/1192

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

CONCURRENT AND FAIL-SAFE REPLICATED SIMULATIONS ON HETEROGENEOUS NETWORKS: AN INTRODUCTION TO ECLIPSE

Felipe Knop
Edward Mascarenhas
Vernon Rego
V. S. Sunderam

CSD-TR-95-014
February 1995

Concurrent and Fail-Safe Replicated Simulations on Heterogeneous Networks: An Introduction to EcliPSe

Felipe Knop **Edward Mascarenhas** **Vernon Rego**
Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907-1398, USA
{knop,edm,rego}@cs.purdue.edu

V. S. Sunderam
Department of Math and Computer Science
Emory University
Atlanta, Georgia 30322, USA
vss@mathcs.emory.edu

Purdue University
Department of Computer Sciences
Technical Report 95-014
February 28, 1995
To appear in: Simulation Practice and Theory

Keywords: concurrent simulation, heterogeneous cluster computation, replication, fault tolerance

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

Abstract

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

1 Introduction

The *EcliPSe* software system was originally designed to support straightforward and semi-automatic concurrent execution of stochastic simulation applications in a variety of parallel and distributed environments [22]. Since its inception, *EcliPSe* has been successful in demonstrating the practical viability of executing replication-based or domain decomposition-based simulations on heterogeneous networks of processors. Indeed, an early prototype demonstrated price-winning performance (a project [18] using the *EcliPSe* prototype was awarded the 1992 Gordon Bell Prize for price-performance by the IEEE Computer Society.) in the investigation of universal constants in a polymer physics application which executed on a country-wide network of 192 processors.

Simulation is known to be computationally intensive, with typical applications often executing for hours or days on fast scalar supercomputers. To reduce execution times, researchers have suggested some techniques for multiprocessor-based simulation. Considerable attention has been given to *distributing* a model over a number of processors in order to speed up the generation of a single sample path, in particular for discrete-event simulation [5]. Examples of this approach include the conservative [17] and the optimistic [8] protocols of distributed simulation. In addition to the complexities of application-level and system-level software development for distributed simulation, performance is often adversely affected by synchronization overheads intrinsic to distributed systems.

An alternate and also complementary approach to model distribution is model replication, which is the approach adopted by the *EcliPSe* toolkit. This fact was already recognized by simulation researchers investigating the statistical consequences of parallel sampling (e.g., see Biles *et al* [1] and Heidelberger [6]). Instead of distributing a single model over n processors, n replications of the same model are made to run on the n distinct processors. This is useful for the most general stochastic simulation paradigm: *several sample paths are required in order that a statement with some statistical basis can be made.* Observe that one cannot avoid replication even for executions based on model distribution.

Thus, model replication may also be used to complement model distribution in that successfully distributed models can be replicated for even better performance. This statement is particularly relevant for large n, because distributed simulation *cannot* guarantee performance improvements with increasing numbers of processors – which suggests that when the performance of a distributed model peaks at $k \ll n$ processors, it makes good sense to replicate the distributed model on the remaining processors. It has been our experience that model replication often exhibits potential for better performance than model distribution simply because replications exhibit few or no data dependencies and do not force synchronization constraints.

The *ACES* project is an effort geared towards producing a software base for simulation applications atop *cluster computing systems* [24], which are systems consisting of heterogeneous networks of workstations and massively parallel hardware multiprocessors. The design of the *ACES* system was motivated by a need for easy experimentation and rapid computation on flexibly configured environments, and its continuing development is guided by the following goals:

ease of use: complex applications should be implementable in a high level manner. A graphics user interface would be an asset in the use of *ACES*.

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

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

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

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

The *ACES* system is organized in layers, as shown in Figure 1. Each of the layers hosts a software system that may be used without knowledge of, or simply without explicit use of the other components in the system. A brief description of each of the component layers is given below.

The lowermost layer hosts a streamlined software base called *Conch* [23], providing higher layers with a virtual multiprocessor machine serviced by an efficient interprocess communications library. Given a set of heterogeneous machines and some user-specified topology, *Conch* builds a powerful multiprocessor environment where processes[1] communicate with the aid of several message-passing primitives. A simple but effective routing mechanism is used to move messages between processes that are not directly connected to each other in the user-supplied topology. *Conch* supports any message-passing based distributed application. Thus, using *Conch* primitives alone, replicated simulation applications can be developed by specifying a set of *slave processes* which compute and return their results to a *master process*, which in turn collects and combines results.

Ariadne is an efficient lightweight process[2] designed to provide support for a variety of concurrency constructs at the *Conch, EcliPSe,* and *Sol* layers. This library allows complex applications to be specified in terms of simpler tasks which can execute concurrently, share CPU time with other tasks by time-slicing, and even migrate between processes executing on different processors to act on data.

EcliPSe adds to the power of *Conch* by facilitating the design and implementation of replicative and domain-decomposed applications, and more importantly, by improving its performance. Writing such an application in *EcliPSe* typically requires less effort than is required using *Conch*. In addition, using a two-level protocol shared with *Conch*, *EcliPSe* supports a robust form of failure-resilient computing at the user-level.

Resident at the uppermost layer, the *Sol* (Simulation Object Library) system facilitates the construction of simulation models and other parallel applications in a variety of domains [2, 3]. *Sol* is a C++ based library, designed to support the *event-scheduling* and *process-interaction* views, based on the object-orient paradigm. *Sol* provides applications with a variety of simulation event calendars and other data structures for the rapid development of simulation models.

[1]A process (e.g. UNIX process) is an executing program with a single thread of control.

[2]A lightweight process is a mini-process with its own stack and local variables. A process may host a large number of lightweight processes.

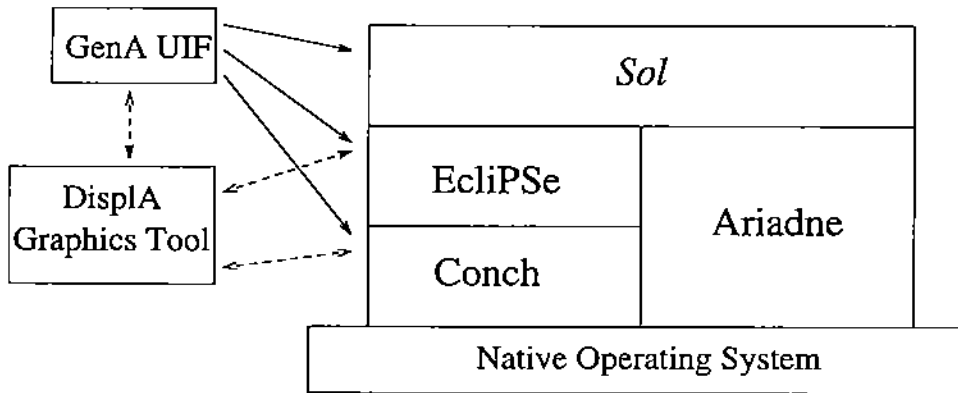

Figure 1: The *ACES* Software Architecture

Finally, the *ACES* system is supported by two useful tools for application development. The *GenA* tool is a user-interface for generating applications at any of the system layers. The *DisplA* tool is a graphical display system that interfaces with an executing application for the purpose of depicting application-related graphical output or performance displays of distributed-system behavior during execution, or for post execution displays of logged data.

The remainder of the paper is focused on the *EcliPSe* layer, which is responsible for providing the base for efficient parallel simulations. The present version of *EcliPSe* is a robust and re-engineered version of the prototype presented in [19, 22], and has already been used in production applications, such as the work described in [20]. A significant feature present in the new system is the capacity for fault-tolerance. Section 2 presents an overview of *EcliPSe*, highlighting the main features. Section 3 presents a brief overview of its fault-tolerance features, and Section 4 describes a performance monitoring tool for *EcliPSe* applications. We present some experimental results in Section 5 and conclude briefly in Section 6.

2 Overview of *EcliPSe*

2.1 Structure

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

An *EcliPSe* program must contain the following components:

Computation code. This is code that is run by each of the *samplers*, being responsible for most of the

"actual work" that the application performs. The computation code usually requires input data from and returns result data to a monitor process.

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

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

2.1.1 Declarations

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

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

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

All "C" basic types (not only `double`) are accepted by the preprocessor. There is also a provision for user-defined types, for which the user must only specify the size in bytes. User-defined types are inconvenient in that *EcliPSe* is not able to provide data coercion for them (when the system is run on heterogeneous machines), since their structure is only known to the application program.

2.1.2 Computation code primitives

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

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

Table 1: Main primitives available in *ElciPSe*. The notion of "parent" and "child" processes is explained in section 2.2.

`request_data` primitives, and (b) replace the collection of result and statistics by the corresponding `put_stat` primitives.

The functionality of the original user code that is replaced by `request_data` and `put_stat` primitives as described above is moved into the monitor function.

2.1.3 Monitor function

All code related to file I/O and to the collection of statistics is placed inside the monitor function, which is executed by a *monitor process*. If a sequential application is being ported to *EcliPSe*, writing the monitor generally means moving the above functionality from the computation code into the monitor. Calls to `produce_data` (counterpart to `request_data`) and `collect_stat` (counterpart to `put_stat`) are inserted where needed. Thus, if the original code reads an input parameter from a file, the *EcliPSe* code will have the file read operation and a call to `produce_data` in the monitor function, and a call to `request_data` in the sampler.

It is worth noting that it is always up to the monitor to decide when the computation must be terminated. The sampler stays in an infinite loop, always working "on demand". Being the only process with some global notion of the computation, the monitor is the sole process capable of deciding when it is time to terminate the computation.

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

2.2 General features

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

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

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

Tree-combining. If a monitor were to receive data directly from a large number of samplers, the amount of incoming traffic and resulting "combining work" could make the monitor a bottleneck for the entire computation. This can happen, for example, when the monitor averages results it receives from samplers. To prevent such a bottleneck from occurring, *EcliPSe* allows processes to be organized in a virtual tree structure, with a user-defined topology and the monitor as the root, as shown in Figure 2. Each sampler transparently sends data to its parent in the tree, instead of sending data directly to the root. Each such parent *combines* its own results with the results it receives from all of its children in the tree, applying the same operation that the monitor process would have applied had the tree-combining scheme not been used. As a result, the monitor at the root only needs to combine data it receives from its own children.

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

Data-diffusing. The virtual tree structure described above can also be used to speed up the distribution of data from a monitor to each sampler. Instead of sending data directly to each sampler, a monitor only needs to use the `produce_data_diffuse` primitive on an array of data items to be distributed. Data is then "diffused" down the tree, with each sampler receiving a data item. The process behaves like the tree-combining scheme in reverse. The same primitive can also be used for an efficient data broadcast. As with tree-combining, the declaration of a data type must be changed to indicate use of data-diffusing. Code for the sampler processes need not change.

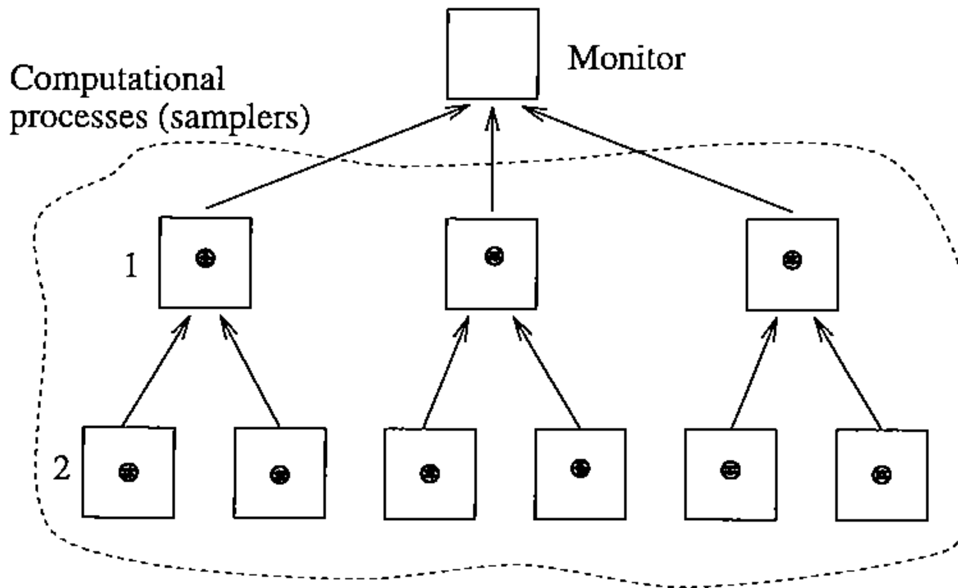

Figure 2: A tree of processes in *ElciPSe*. The dotted boxes are sampler processes that compute and also perform combination of results. Process 1 is process 2's parent in the tree. The arrows indicate the direction of data flow in a tree-combining operation.

Figure 3 shows the outline of a parallel replication-based simulation program built on top of *EcliPSe*. The example corresponds to an M/GI/1 queue[3] simulator. Upon receiving a distinct random number seed, each sampler proceeds to simulate the queue for a certain number of customers, generating a sample of the average delay on the system and of the maximum number of elements in the queue. The samples are accumulated at the monitor, which stops the computation when a confidence interval is obtained. Although the *EcliPSe* primitives and declarations shown in Figure 3 correspond to an M/GI/1 queue simulator, the program's structure changes little for other replication-based applications. In the example of Figure 3, the input parameters are broadcast and seeds are "diffused" from the monitor to all samplers. The two simulation outputs (average delay on system and maximum number of customers in queue) are sent to the monitor with the tree-combining mechanism. The "grain size" mechanism is also used in the example. In Figure 3, function `accum_collect_stat_combine` is an *EcliPSe* primitive built on top of `collect_stat_combine`. It combines the (combined) results from all samplers with the accumulated results from previous calls to `accum_collect_stat_combine`.

2.3 The GenA User Interface: Auto Monitor Generation

One of the goals of the *ACES* project is ease of system use. With a good interface, a user is able to learn to use as well as adapt his application to a new software system rapidly. Towards this end we have developed *GenA*, a user interface that provides an application development and execution platform for the *ACES* system. This interface is targeted towards users who are familiar with an application domain but

[3] An M/GI/1 queue is a single server queueing system with Poisson arrivals and general, independent service times.

```
main()
{
    Variable declaration, initialization
    eclipse_decls{
        int type_seed(inputdata diffuse);
        double type_interarr_time(inputdata diffuse_same);
        double type_srv_time(inputdata diffuse_same);
        double type_delsys(combine tree avg, grain_size 8);
        int type_maxqueue(combine tree avg, grain_size 8);
    }
    (From this point, only computational processes execute the code below. The monitor starts executing
     the monitor() function)
    computation();
}

monitor()
{
    Read parameters and seeds array from file
    produce_data_diffuse(type_interarr_time, &interarr_time);
    produce_data_diffuse(type_srv_time, &srv_time);
    produce_data_diffuse(type_seed, seed_array);
    while(confidence interval not reached){
        Receive combined data from all computational processes and combine with accumulated results:
        accum_collect_stat_combine(type_delsys);
        accum_collect_stat_combine(type_maxqueue);
        Compute statistics
        Evaluate termination criterion
    }
}

computation()
{
    Receive parameters and seed from monitor:
    request_data(type_interarr_time, &interarr_time);
    request_data(type_srv_time, &srv_time);
    request_data(type_seed, &seed);
    while(forever){
        Simulate M/GI/1 queue batch
        put_stat(type_delsys, &avg_delay_on_system);
        put_stat(type_maxqueue, &max_elements_on_queue);
    }
}
```

Figure 3: M/GI/1 example in *EcliPSe*.

need help in organizing and managing their applications during system use. For example, *GenA* places source files pertaining to a single project within the same directory, calls the correct compiler on each of the user's source files, and creates an executable linked with all necessary libraries.

GenA provides support for running applications in batch mode and can even generate source code for the *monitor* function in *EcliPSe* applications. The part of the *GenA* interface that generates this code is called *EclGen*. This code generator is very useful in the parallelization of sequential applications. It is typically the case that a user has a working sequential program which is to be parallelized (i.e., replicated or domain-decomposed) using *EcliPSe*. Such a user would need to modify his `main()` function, and add a `monitor()` function to the code. The *EclGen* tool does most of this work: it generates all necessary code to initialize *EcliPSe* as well as code for the monitor. The user is expected to interface existing code correctly with the automatically generated monitor. We have found the graphical user interface to be useful for *EcliPSe* applications. We next describe the *EclGen* interface, the monitor code generator, and features of the automatically generated code.

2.3.1 *EcliPSe* Code Generator

EclGen is an X/Motif [7] based tool that makes use of menus, dialogs, and direct manipulation (point and click) to input an *EcliPSe* specification, and then generates the *EcliPSe* data type declarations code and the monitor code that handles those data types. It allows the use of a high level description of the monitor to generate the final source code for the monitor. It is a user directed interface in the sense that the user fills in a template describing the monitor and its data types; the system then generates the required code. Interfacing the remainder of the application with the code generated by *EclGen* is simple for *EcliPSe* applications – mainly because these applications are well-structured and belong to a set of basic forms [10].

An example structure for an M/GI/1 application is shown in Figure 3. In this figure the `eclipse_decls` portion of the `main()` function and the `monitor()` function are completely generated by *EclGen*. The user is left with the task of writing other parts of the `main()` function and the `computation()` function. The *EclGen* software system consists of two parts. The user interacts with the front-end which accepts a specification. This specification is then used by *EclGen* internals to generate code. This occurs on user command, by clicking on a menu option. The code generator reads in the specification and makes use of a lexical analyzer and parser to convert the specification to C code.

A user specification consists of the monitor description and a list of data types and their details, filled out in the template shown in Figure 4. The monitor description part specifies the monitor identifier (integers starting at zero), whether data conversion to hardware independent form for data sent over the network is required, and the monitor function name. For *EcliPSe* applications the monitor must contain code that determines when the computation should terminate. This termination checking function, which is either a user-written function or an *EcliPSe* (built-in) function, is also specified here. Several built-in functions are available to aid in different types of *EcliPSe* applications (see [10]). The input data file name for the distinct data types read in must also be specified. They may be used to input simulation parameters at run time. Details for each data type include the name, type, diffuse or combine, and grainsize. The

Monitor Id: 0 Monitor Function: monitor_0

Save to File: Data Input File:

Data Convert NO | Seed Input User | Termination User defined | terminate_0

Function Add | Type C_INT | Diffuse DEFAULT

Combine DEFAULT | CombineOp CO_ADD

Name	Type	Elements	Varname	Diffuse	Combine	Combir
Seed	C_INT	1	seed	DIFFUSE		
Inter arrival t	C_DOUBLE	1	interarr_time	DIFFUSE_SAME		
Service time	C_DOUBLE	1	srv_time	DIFFUSE_SAME		
Delay in system	C_DOUBLE	1	delsys		TREE	CO_A

Save LoadFile SaveClose Cancel ClearAll Help

Figure 4: EcliPSe Auto Generator Template

Diffuse and **Combine** information is used by the code generator to determine whether the data type is an input data type or an output data type.

Three files are generated by the code generator. All declarations of generated functions and *EcliPSe* user data types are written to a `.h` file; input data read routines, and code to handle the monitor functions are placed in a `.c` file. A `_user.c` file is made to contain stubs for functions which the user may want to fill in or rewrite if the default action is not the desired one. For example, the default termination check allows each sampler to send just one sample. The user may provide another function to override this behavior, if necessary. Other stubs included by default in this file are a function to do some application specific initialization when the monitor begins to execute, and a function that is called when the monitor finishes. The latter function may be used to print results from the computation. Retaining these stub functions in the code without modification does not affect the computation in any manner.

The generated code has several aspects that are worth mentioning. All `produce_data()` and `collect_stat()` associated data types are bundled together into a single, global C data structure called `struct_monitor_0` for monitor 0 (if there is more than one monitor, several such structures will be generated). These correspond to the *EcliPSe* user defined data types. Values read from the input data are stored in this structure before these are sent to the samplers. Results of the computations are also stored here. Because the structure is global, it can be accessed by user written functions. For example, a function to summarize and print results may access this structure.

Routines generated to read in input data from files must read the correct number of values depending on the number of processes specified in the runtime configuration. Whether one value is read or multiple values are read depends on whether the data type is **DIFFUSE** or **DIFFUSE_SAME**. The generated function `eclipse_options()` sets all the *EcliPSe* options and defines all *EcliPSe* data types. The monitor function itself includes the calls to `produce_data()` for each **Diffuse** data type, the call for termination check, and the `collect_stat()` calls for each **Combine** data type after every termination check. The termination check function can be provided by the user. For termination policies like *auto-scheduling* until all work is completed, until a specified *confidence level* is obtained (see [10]) or some other application-specific policy, the users must supply a termination function. Code for timing information is also generated automatically.

An example of the use of *EclGen* in automating monitor generation for the M/GI/1 queue simulator introduced in Section 2.2 is presented in Appendix A.

3 Fault Tolerance

Typical *EcliPSe* applications often require large computational resources, sometimes implying the use of tens or hundreds of machines executing continuously for several days, or possibly weeks. In this setting, the need for fault tolerance is critical, because heterogeneous distributed computing in an open, uncontrolled environment is generally unreliable for a number of reasons. Failures may be due to alien processes sharing one or more processors with the distributed computation, independent workstation reboots by station users, or simply hardware failures. If no fault tolerance is provided, long-running applications may never run to completion.

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

1. **Process or machine failure**. In general these are caused by operating system resource exhaustion (caused by an *EcliPSe* application or an alien application using one of the *EcliPSe* machines), operating system error, hardware fault, machine shutdown, or network failure.

   The default action in each case is to run a replacement process that substitutes for the failed process.

2. **Software exceptions**. These are exceptions detected by the hardware/operating system and are usually due to application programming errors. Executing a replacement process is not wise since the error is likely to repeat itself unless corrected by the application programmer. Only an appropriate error message is printed and the application is typically terminated.

3. **Infinite loops in the application.** Even if the application successfully passes small-scale test runs, problems may arise with large-scale runs. A problem that might occur is that one or more samplers enters an infinite loop because of a subtle programming error at the application level. Alternately, a sampler may appear to be in an infinite loop if it works slowly, due to a hitherto undetected load surge on its host machine or because it has been given too low a priority by its host's scheduler.

A hard problem to tackle in the general case, detection of infinite loops is performed using some user-guided heuristics. Upon detection of such a situation, appropriate user action is taken, with the default being simply a warning message.

To allow recovery from a process or machine failure, a checkpoint-rollback mechanism is used: data from all processes is periodically saved, and then later restored should a failure occur, with a new process being created to replace one which fails. Some cluster-computing systems provide transparent checkpoint and recovery, in the sense that no programming effort is required [4, 14, 21]. The drawback of this approach is a heavy checkpoint overhead, since the complete address space of all processes involved must be saved at each checkpoint. While *EcliPSe* checkpointing is not transparent to the application, it requires little programming effort and has the added advantage of being low cost and efficient; the user only needs to declare what must be saved and specify the few points in the program where checkpoints should occur. To specify the data to be saved, the user must declare a set of *recovery data types* together with a set of data pointers. As an example, a 20-element integer state vector is saved by all samplers at a checkpoint, and then restored in a rollback, with the following declaration:

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

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

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

Provided adequate state vectors are specified, rollbacks still allow the program to produce the same output regardless of the number of process failures even when samplers base their computations on random numbers. For this, a notion of global system state is maintained, with the global state being restored by the monitor and samplers after a process failure. All data messages in transit or in buffers at the time of the failure are automatically discarded by the system.

The heuristics used for infinite loop detection are based on the processes' past history. *EcliPSe* records the interval between two subsequent data messages sent by each process to the monitor (or to the process's parent if the tree-combining mechanism is used). If a process stops sending data for a period that is much greater than either its previous maximum interval or the combined maximum interval of all processes, the process is considered to be in an infinite loop. Based on the application's structure and on the execution environment's characteristics, the user may specify multipliers for the previous maximum intervals, to account for some normal variations in the sample generation times. Large multipliers must be specified when the application is executed on machines with unpredictable loads or for programs where the time to generate a sample is highly data-dependent.

A shutdown-restart mechanism has also been provided: the user may stop the application and restart it later, possibly using another set of machines (even of a different type). Shutdown can be invoked if the user knows in advance that a large fraction of the machines in use is becoming unavailable due to

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

Table 2: Performance statistics collected by the tool

an imminent network partition. By taking advantage of the application's structure, *EcliPSe* programs can often achieve an almost negligible checkpoint overhead, which in turn drastically reduces the fault tolerance performance penalty. The details of how this is accomplished, as well as a more in-depth description of fault tolerance in *EcliPSe*, are presented in [13].

4 Performance Measurement

EcliPSe applications support a variety of computation structures (see [11]) and execute on a number of machine environments. Bottlenecks in a distributed application, however, may impair execution performance, resulting in a waste of computational resources. Early experience with some *EcliPSe* applications indicate that bottlenecks tend to occur when (a) the monitor is overworked by a high influx of data messages, and (b) samplers have to share a host CPU with other non-*EcliPSe* applications. Some bottlenecks are easily circumvented by a small change in the program or execution environment, provided the source of the bottleneck can be found.

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

As an example of the tool's usage, suppose that histograms reveal the monitor's CPU to be occupied at the 40% level, with a load average of 3.05. We may readily conclude that processes (possibly alien

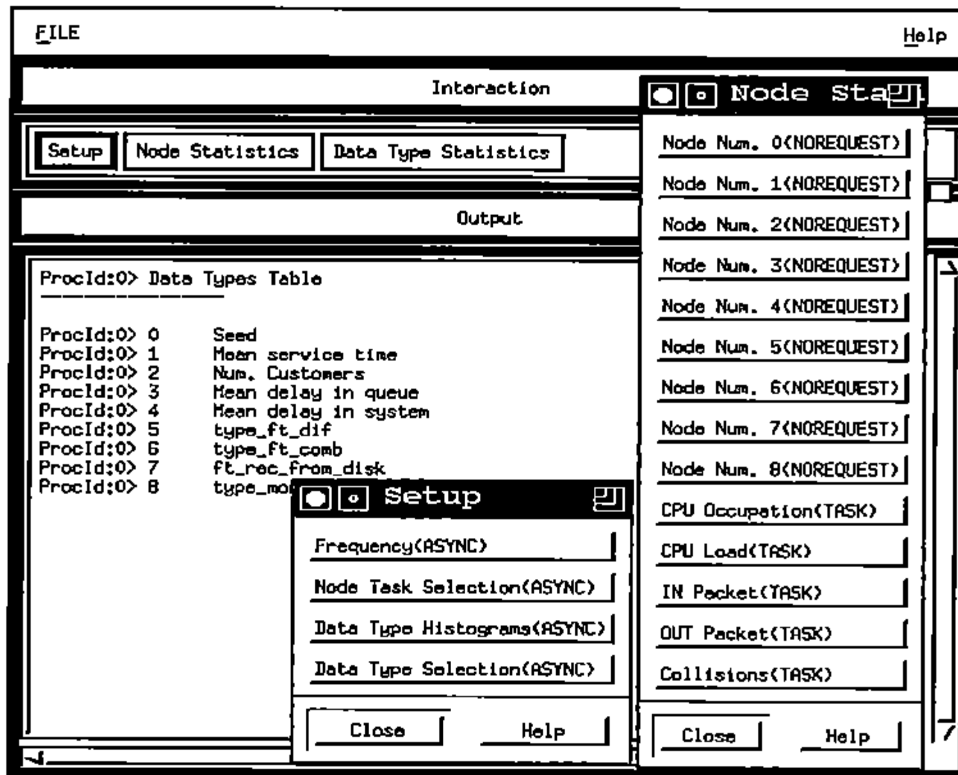

Figure 5: Main Window of Display Tool

to *EcliPSe*, if the user has not loaded more than a single *EcliPSe* process on this CPU) competing for attention from the host CPU prevent the monitor from executing as fast as it could, had it been executing on a dedicated CPU. With a dedicated CPU, the monitor's CPU occupation level would rise to almost 100%, and the distributed application would execute faster. We should also conclude that a computational bottleneck occurs at the monitor, since the occupation-level and load average combination indicate that it is likely to be using all the CPU attention available to it, and is probably not able to process incoming messages at the required rate. This problem can be resolved immediately by changing the program or the execution environment, or at least relocating the monitor to a lightly loaded CPU. If, for the same CPU occupation level (40%), the load average was found to be 0.4, we may conclude that the monitor is being hosted by a dedicated CPU and also that it is able to cope with its incoming messages and combining work.

The main window of the tool is shown in Figure 5. The buttons shown under the title *Interaction* allow a user to interact with an ongoing *EcliPSe* application. Dialogs can be used to change the frequency of performance information updates, input data to the application, and to enable the display of some desired output. The *DisplA* library provides a user with a simple interface, allowing for calls that can create application specific dialogs and receive input from the user [15]. Using the *Setup* panel and clicking on options available, a user can select/deselect the plotting of histograms for CPU occupation-level, CPU load, in/out packet rate, etc.

To enable *DisplA*, a user simply needs to link the application with the *DisplA* library. *DisplA* executes as a server on the initiating workstation. Once initialized, *EcliPSe* is able to determine whether a *DisplA* server is present and connect to it. When the user inputs a nonzero frequency to the server, it begins to obtain performance-data updates and displays the information requested at the specified frequency. If required, display can be done on a specific workstation, including one that is distinct from the workstation which initiates the application. Facilities are also available to log performance-data so that post-execution performance graphs can be displayed.

5 Experiments

To give the user an idea of how *EcliPSe* performs on easily understandable examples, we present the results of a few experiments. One example involves replications of an M/GI/1 queuing process, another involves the estimation of a multidimensional integral and a third involves replications of a random walk on a finite state space. The goal is to demonstrate the utility of some of the main features described in the paper, and to give some indication of *EcliPSe*'s performance on typical applications with different computation structures. Another set of experiments is reported in [12].

5.1 Machines

The experiments were conducted on a network of 177 SUN IPC SPARC workstations located at the ENAD laboratories at Purdue University. These machines reside on four distinct local area networks, all connected via a single gateway. The communication time (as measured by the UNIX `traceroute` command working with 40-byte packets) between local machines is 2 ms, and this increases to 3 ms when a message has to pass through the gateway. Each machine is rated at 15 MIPS. Though the machines were primarily dedicated to our *EcliPSe* applications at the time the experiments were performed, daemon processes were frequently observed running (i.e. competing with *EcliPSe* processes), sometimes imposing a non-trivial load on the machines.

5.2 Programs

Though the experiments were performed using very simple applications, namely the M/GI/1 queue, integral estimation, and the random walk, all three are easily generalizable representations of typical replicative applications in *EcliPSe*. Examples of domain-decomposed applications are given in [12].

5.2.1 M/GI/1 queue simulation: replication

In this program (hereafter referred to as **mg1**), each replication simulates an M/GI/1 queue for a fixed number of arrivals. Based on batch-means, the statistics sent to the monitor include mean system delay and maximum number of customers found in the queue. The regenerative method may also be used (as was done in [19]) to estimate these quantities. The monitor utilizes results from independent parallel

replications to build a confidence interval for both statistics. Though times between the reporting of samples can be large, the monitor can be overworked if a large number of processes is used. For the experiments reported in this paper, the total number of samples collected was fixed. Two variations of the program were tested: one *with* and one *without* tree-combining.

5.2.2 Multidimensional integral estimation

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

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

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

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

5.2.3 Absorption times in Markov chains

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

The parameters used in this experiment were $k = 256$ and `grainsize` $= 128$. A total of $2^{18} = 262144$ samples are generated in all by the samplers. A fault-tolerant version of the application was used, based on the fault tolerance interface outlined in Section 3. For fault-tolerance, a state size of 8 bytes was used for each sampler, with 36 bytes for the monitor. Although one call to `check_recover()` is executed for each sample generated, the actual saving of checkpoint data is only made every `grainsize` samples. Upon failure of a process, matrix P is sent to the replacement process where a new sampler resumes execution from a state just prior to the failed sampler's state at the time of failure.

5.3 Measurements

Each time an *EcliPSe* application is run, a virtual machine environment must be built. This entails executing a *single* process on each machine and setting up all connections. This procedure is somewhat time consuming when the number of processes is large (around 3 minutes for 64 processes). Nevertheless, this start-up time was not considered in the measurements, since it is believed that typical applications

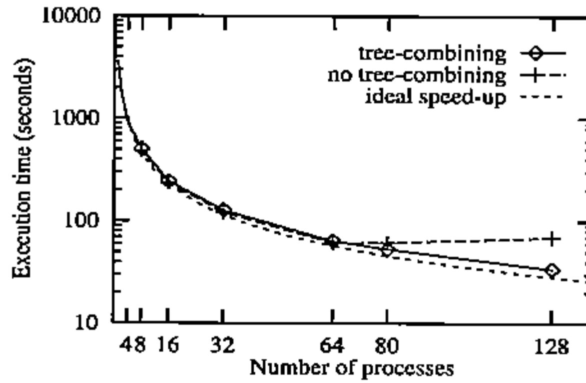

Figure 6: Execution times for M/GI/1 application

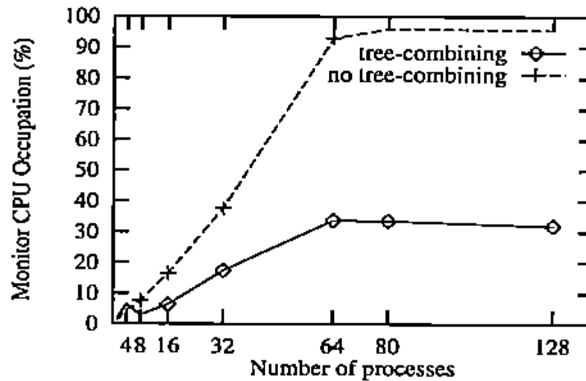

Figure 7: Monitor CPU occupation level for M/GI/1 application

will take a long time to run, making the start-up time negligible. With the advent of high speed networks and improvements to *Conch*, we expect this time to decrease.

5.4 Results

5.4.1 Effect of the tree-combining scheme

Program **mg1** was chosen for the evaluation of the tree-combining scheme. In the version without tree-combining, samples of average delay on the system and samples of maximum number of customers in queue are sent directly to the monitor, which utilizes a running-mean procedure to compute sample mean and variance. In the tree-combining version, the **average** combining operation is used, together with the *EcliPSe* monitor function which accumulates results. For both programs, 25000 customers were simulated in each replication, for a total of 2048 replications. Figure 6 shows the execution time for both versions of the program, while Figure 7 shows the corresponding CPU occupation-level of the monitor.

For fewer than 64 processes, the "normal" combining program was slightly (up to 5%) faster. This happens because of combining overhead imposed on the samplers and also because of delay incurred by samples arriving at the monitor (as explained in [12]). For a larger number of processes, the "normal"

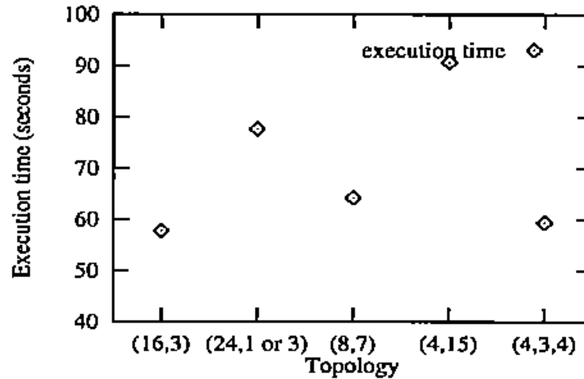

Figure 8: Execution times for the different tree topologies

combining program was not able to take advantage of the extra samplers, and an explanation for this can be found in Figure 7: the monitor is overloaded with work generated by a high influx of messages. As an aside, 70% of the monitor CPU time was used in *system* mode, probably caused by operating system overhead (i.e., protocol and system-level handling of incoming network packets).

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

5.4.2 Effect of different tree topologies

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

To describe each tree topology, the following notation is used. A $(x_1, x_2, \ldots)$-tree is a tree where the root has x_1 children, each of which has x_2 children, and so on. Figure 8 shows the program execution times for the following topologies: $(16, 3)$, $(24, 1\ or\ 3)$ (in this topology, each of the 24 nodes in the first level has either 1 or 3 children, so that the total number of nodes is still 64.), $(8, 7)$, $(4, 15)$, and $(4, 3, 4)$. Figure 9 shows the corresponding CPU occupation-level of the monitor.

The values of the monitor's CPU occupation-level are consistent: the load of the monitor tends to be proportional to the number of children it has. Using the limited data provided by Figure 8, we may arrive at the following conclusions: (a) tree topologies that overwork the monitor (like $(24, 1\ or\ 3)$ for the present experiment) result in poor speedup, and (b) tree topologies that avoid overworking the monitor at the expense of assigning several child processes to each non-monitor process (like $(4, 15)$) also behave poorly.

Figure 8 indicates that $(16, 3)$ and $(4, 3, 4)$ are the best topologies for the programs and parameters used in this experiment. If we were to choose between the two, the latter would certainly be selected if tasks such as the graphical displaying of data or other potentially compute-intensive tasks were to be added to the monitor, since $(4, 3, 4)$ imposes a much lighter load upon the monitor process.

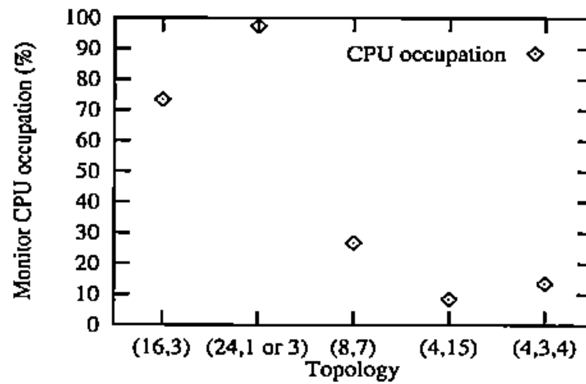

Figure 9: CPU occupation-level of Monitor for different tree topologies

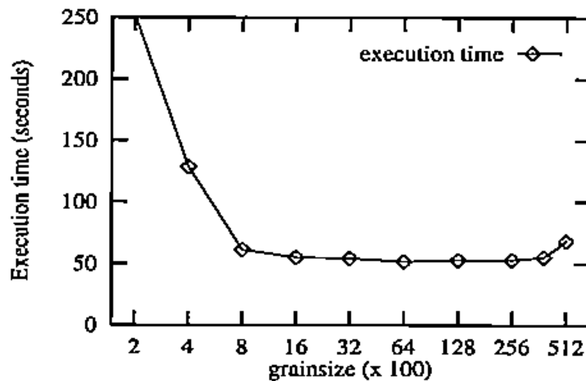

Figure 10: Execution times for the different values of grainsize

5.4.3 Effect of the grainsize

The program chosen for the "grainsize" experiment was **integral**, using the tree-combining scheme. The total number of samples was 8192000, and the program was run with 64 processes placed in a $(16, 3)$-tree. Figure 10 shows the program's execution times for grainsize varying from 100 to 51200, while Figure 11 shows the corresponding monitor occupation-level of the CPU.

For a grainsize smaller than 800 the program's execution time increases roughly proportionally with the inverse of the grainsize, which can be attributed to the monitor not being able to handle the incoming data. Data in Figure 11 indicates that with grainsize 800 the monitor is using almost all CPU time available, and that the total amount of work accomplished will only decrease if the grainsize is made less than about 800.

For a grainsize larger than 800 the monitor load is much less of a factor, and ordinarily the program's execution time should decrease until an asymptote is reached. An explanation of the unexpected increase in execution time for a very large grainsize may be found in the program's pattern of memory usage: a larger grainsize requires a large amount of memory to be used by the samplers to store samples, therefore causing an increased number of page faults. This problem is further aggravated both by sample buffering and sample combining. Both use memory in a sequential manner, causing cache and memory replacement

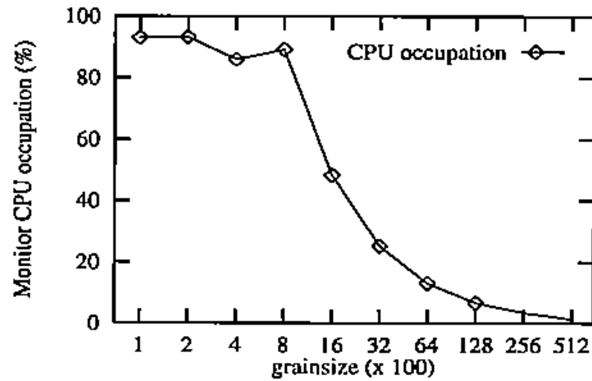

Figure 11: Monitor CPU occupation for the different values of grainsize

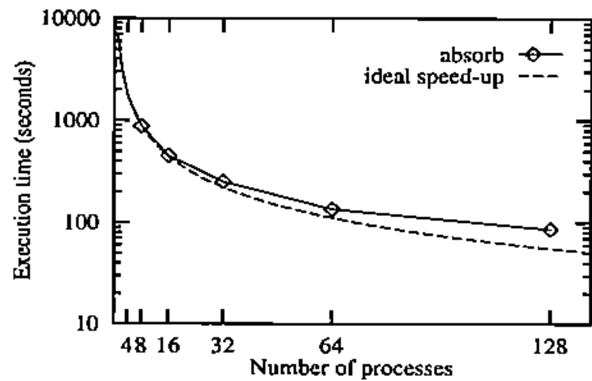

Figure 12: Execution time for the **absorb** program

algorithms to exhibit poor performance. When grainsize is used with the tree-combining scheme (as in the present experiment), a solution to this problem would be for `put_stat` to combine the process's samples every time X new samples are obtained, when a large grainsize is being employed. This would lessen sample buffering problems, at the expense of having more frequent combine operations within samplers.

In spite of the slight increase in execution time for large values of grainsize, this experiment demonstrates that the grainsize mechanism is a powerful way of controlling the monitor's load.

5.4.4 Fault-tolerance overhead

Program **absorb** was run with different numbers of processes to evaluate the effects of checkpointing overhead. Execution times are shown in Figure 12. From this figure it is clear that the application is still able to obtain good speed-up, despite frequent checkpoints and the significant proportion of time (about 20% of the total execution time for 128 processes) spent by the monitor to broadcast the transition probability matrix. This graph shows that fault-tolerant applications in *EcliPSe* can be developed with negligible checkpoint overhead, using low programming effort to make the program fail-safe.

6 Conclusions

Our experiments with the *EcliPSe* system have led to interesting results. Simple fault-tolerant applications can be designed and made to run on a network of dozens of machines in a matter of a few hours. Such programs are fairly easy to understand and modify, and perform very well in a multi-machine run. This indicates the effectiveness of *EcliPSe* in speeding up the execution of model replication applications. In a run which was larger than those used for performance evaluation, the **mg1** program was executed on 160 samplers, being able to complete 32800 replications of a 100000-customer simulation in about 26 minutes.

Working with *EcliPSe* also gives us a feel for possible improvements. This system has been designed around the idea of having one or more monitors coordinate a potentially large number of samplers. In general, the modifications required (relative to an already existing sequential version) to computational code are simple, requiring only the insertion of some `request_data` and `put_stat` calls. However, the monitor code tends to be a little more complicated, with most of it being similar across different applications possessing similar computation structures. This has prompted us to develop and enhance the *EcliPSe* code generator described in Section 2.3.

We now evaluate the ACES system in regard to its goals, set in Section 1.

The ease of use of ACES has been demonstrated by several applications developed or ported to the system. Replication-based parallel simulation programs have been adapted for *EcliPSe* in a matter of hours. Complex discrete-event simulations have been built using the high-level constructs provided by *Sol*. The development of applications in the ACES system is also facilitated by the *GenA* graphical user interface.

The ACES system has been ported to several machine architectures, such as heterogeneous networks of workstations, including SUN SPARCstations and IBM RS6000s, and shared memory multiprocessors (Sun SPARCCenter 1000, Sequent Symmetry). The system is currently being ported for the Intel Paragon, a distributed memory multiprocessor. *EcliPSe* allows all those machines to be used together in a single run, since data is exchanged between processors in a network independent format (thanks to the data declarations described in Section 2.1.1). The porting of ACES to different architectures is simplified by system layering, since only low-level *Conch* and *Ariadne* functions need to be ported to new architectures.

The distinct layers of the ACES system provide functionalities that suit several different simulation programs. *Sol* and *Ariadne* provide constructs for process-based simulation, *Conch* supports generic message-passing distributed programs, and *EcliPSe* has constructs geared towards replication programs, but also supports more general forms of parallel applications such as that described in [9].

EcliPSe's features such as the tree-combining, data diffusing, multiple monitors, and granularity control are essential in avoiding serializing bottlenecks and therefore allow better scalability. Also essential in achieving good scalability are *Conch*'s support for different interconnection schemes, which take advantage of the communication topology to reduce communication times.

Fault tolerance, an essential feature especially for long-running simulation programs, is incorporated at the *Conch* and *EcliPSe* layers. *Conch* provides the basic support for process replacement: a process that fails is replaced by a new one, possibly in a different machine. *EcliPSe* builds on top of *Conch* to

provide the fault tolerance features described in Section 3.

Finally, our experiences and empirical results with *EcliPSe* indicate that automating the virtual machine configuration procedure may result in improved application performance. A tool is presently being designed to generate a process topology which is optimized for (a) machine power and load, (b) network topology and traffic, and (c) characteristics of the application.

References

[1] B. Biles, D. Daniels, and T. O'Donnell, Statistical considerations in simulation on a network of microcomputers, in: *Proceedings of the Winter Simulation Conference*, San Francisco, CA (1985) 388–393.

[2] K. Chung, A Concurrent Composite Computational Model for Stochastic Simulation, Ph.D. Thesis, Purdue University, West Lafayette, IN, 1993.

[3] K. Chung, J. Sang, and V. Rego, *Sol-es:* An object-oriented platform for event-scheduled simulations, in *Proceedings of The Summer Simulation Conference*, Boston, MA, (1993) 972–977.

[4] H. Clark and B. McMillin, DAWGS – a distributed compute server utilizing idle workstations. *Journal of parallel and distributed computing* **14** (2) (1992) 175–186.

[5] R.M. Fujimoto, Parallel discrete event simulation, *Communications of the ACM*, **33** (10) (1990) 30–53.

[6] P. Heidelberger, Discrete event simulations and parallel processing: statistical properties, *SIAM Journal on Scientific and Statistical Computing*, **9** (6) (1988) 1114–1132.

[7] D. Heller and P. M. Ferguson, *Motif Programming Manual for OSF/Motif release 1.2* (O'Reilly & Associates, Sebastopol, CA, 1994).

[8] D. Jefferson, Virtual time, *ACM Transactions on Programming Languages and Systems*, **7** (3) (1985) 404–425.

[9] F. Knop and V. Rego, Parallel cluster labeling in a network of workstations, Submitted for publication, Purdue University, West Lafayette, IN, 1995.

[10] F. Knop, V. Rego, and V. Sunderam, *EcliPSe* User Manual. Technical report, Purdue University, West Lafayette, IN, 1993.

[11] F. Knop, V. Rego, and V. Sunderam, *EcliPSe*: A system for fault-tolerant replicative computations, in *Proceedings of the IEEE/USP International Symposium on High-Performance Computing*, São Paulo, Brazil (1994) 17–34.

[12] F. Knop, V. Rego, and V. Sunderam, *EcliPSe*: a system for parallel execution of replication-oriented applications, In preparation, Purdue University, West Lafayette, IN.

[13] F. Knop, V. Rego, V. Sunderam, and A. Ferrari, Failure-resilient computations in the *EcliPSe* system, in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL (1994) III-184–III-187.

[14] J. León, A.L. Fisher, and P. Steenkiste, Fail-safe PVM: a portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1993.

[15] E. Mascarenhas and V. Rego, *DisplA*: A Graphical Display Server for Concurrent Computations, In preparation, Technical report, Purdue University, West Lafayette, IN, 1994.

[16] E. Mascarenhas and V. Rego, *GenA*: A GUI for Generation of *ACES* Applications, In preparation, Technical report, Purdue University, West Lafayette, IN.

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

[18] H. Nakanishi, V. Rego, and V. Sunderam, Superconcurrent simulation of polymer chains on heterogeneous networks, *1992 Gordon Bell Prize Paper* in: *Proceedings of the Fifth High-Performance Computing and Communications Conference: Supercomputing '92*, Minneapolis, MN (1992) 561–569.

[19] V. J. Rego and V. S. Sunderam, Experiments in Concurrent Stochastic Simulation: The EcliPSe Paradigm. *Journal of Parallel and Distributed Computing* **14** (1) (1992) 66–84.

[20] M.D. Rintoul, J. Moon, and H. Nakanishi, Statistics of self-avoiding walks on randomly diluted lattice, *Phys. Rev. E* **49** (1994) 2790–2803.

[21] G.C. Shoja, A distributed facility for load sharing and parallel processing among workstations. *Journal of systems and software*, **14** (3) (1991) 163–172.

[22] V. S. Sunderam and V. J. Rego, *EcliPSe*: A system for High Performance Concurrent Simulation, *Software-Practice and Experience*, **21** (11) (1991) 1189–1219.

[23] B. Topol, Conch: Second generation heterogeneous computing, Technical report, Master thesis, Department of Math and Computer Science, Emory University, Atlanta, GA, 1992.

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

APPENDIX

A Auto Monitor Generation: Example of Interface Use

We demonstrate the use of the interface with the M/GI/1 example of Figure 3. The user fills out the template shown in Figure 4 on his workstation. The template contents are saved and can later be reloaded and changed. For this application, five *EcliPSe* user data types were defined. These become part of the `struct_monitor_0` structure shown in Figure 13. The first data type is the `seed` (the user can make use of the built-in seeds for random numbers from the *EcliPSe* library or provide them in the input data file.). The next two are input parameters, with values read in from the input file. These are the mean inter-arrival time (`interarr_time`) and the mean service time (`srv_time`). The remaining two data types are results to be computed by the samplers: delay in the system (`delsys`), and the maximum queue length (`maxqueue`). The code generated by *EclGen* in this case includes the `eclipse_options()` function, the functions to read in input data, and the `monitor_0()` function. *EclGen*-generated routines to read in input data, namely the values of seed, the inter-arrival time, and the service time did not require any modification. Also, the termination check function that was generated was not changed in this example, because the computation was terminated after simulation of a fixed number of customer services. Use of the regenerative method (see [19], for example) would require a change in this function.

An existing sequential program was used for the example. The sequential code only required the writing of a new `main()` function to call the `eclipse_options()` function and the `computation()` function. Thus, with very few changes, an existing sequential simulation program could be converted to be used with *EcliPSe*. A sample of the code generated for the monitor is shown in Figure 14. The original `computation()` required changes for inclusion of `request_data()` and the `put_stat()` primitives, so that the computation code can exchange data with the monitor. The complete code is provided in [16], along with other detailed examples on the use of the interface.

```
struct struct_monitor_0 {
    int seed[MAXPROCS]; /* MAXPROCS describes the cluster size */
    double interarr_time;
    double srv_time;
    struct double_avg *accum_delsys;
    int *accum_maxqueue;
} rec_monitor_0;
```

Figure 13: Function interface structure for M/GI/1 example

```
/* monitor_0() - monitor function */
int monitor_0()
{
    int i,j;
    char str[256];
    double bef, aft, msecond();

    open_monitor_0();
    read_monitor_0(&rec_monitor_0);
    bef = msecond();
    produce_data_diffuse(type_seed,
        (char *)&(rec_monitor_0.seed));
    produce_data_diffuse(type_interarr_time,
        (char *)&(rec_monitor_0.interarr_time));
    produce_data_diffuse(type_srv_time,
        (char *)&(rec_monitor_0.srv_time));
    initialize_monitor_0(&rec_monitor_0);
    termination();
    aft = msecond();
    sprintf(str, "Time elapsed = %f\n", aft-bef);
    e_print(str);
    results_monitor_0(&rec_monitor_0);
}

void termination()
{
    int pr;
    int retval;

    (char *)rec_monitor_0.accum_delsys =
        accum_collect_stat_buf(type_delsys);
    (char *)rec_monitor_0.accum_maxqueue =
        accum_collect_stat_buf(type_maxqueue);

    while (terminate_0() == NO) {
        accum_collect_stat_combine(type_delsys);
        accum_collect_stat_combine(type_maxqueue);
    }
}
```

Figure 14: Auto-Generated Monitor: M/GI/1 Example (error checking deleted for brevity)