# On the Effectiveness of Superconcurrent Computations on Heterogeneous Networks*

Hisao Nakanishi
Department of Physics
Purdue University
West Lafayette, IN 47907

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

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

**Abstract**

Concurrent computing on networked collections of computer systems is rapidly evolving into a viable technology that is attractive from the economic, performance, and availability perspectives. Several software infrastructures that support such heterogeneous network-based concurrent computing have evolved, and are in use for production-quality high-performance computing. In this paper, we describe such a system, and present our experiences with its use for massively concurrent computing in the application domain of polymer physics. The application involves stochastic simulation of polymer chains for measuring scale-invariant phenomena at critical disorder. The parallelization is achieved through the EcliPSe toolkit, and conducted on a flexible, tree-structured virtual machine made up of arbitrary and heterogeneous computing nodes dispersed across the country. These nodes cooperate to perform the simulation and pool results together in real time at a central node which initiates the parallel simulation. The advantage of the tree-structure is that it allows for a fault-resilient, flexible environment for long-runing parallel simulations. We present results on sheer performance, price-performance, and toolkit-based parallelization by monitoring the same computations on a CRAY Y-MP and making detailed comparisons. Most significant are the excellent price-performance ratios given by the heterogeneous computing environment.

*This project was awarded the 1992 Gordon Bell Prize for price-performance. An abridged version of this paper was presented at Supercomputing '92 in Minneapolis, MN. This research is currently supported by the NSF, DOE, ONR, ARO and NASA.

# 1 Introduction

Parallel or concurrent computing is now widely accepted as inevitable for high-performance computing. However, widespread parallel processing is hindered by two serious obstacles—the difficulty of parallel software development, and the limited access to, changing architectures of, and high cost of hardware parallel processors. In a recent commentary on the (limited) usefulness of today's parallel computers, Bailey [1] makes the point that a high level of sustained performance on real scientific applications, coded with reasonable effort, is the most important consideration in a highly parallel computer. He also suggests that if parallel computers are ever to be widely used, they should be capable of handling a large number of users in daytime production hours.

One effective strategy that partially addresses the above drawbacks is the use of concurrent computing environments or software systems that emulate parallel machines over a networked collection of heterogeneous computer systems. This approach is viable for a number of reasons, including availability, stability, cost-effectiveness, and ease of use. Several systems based on this concept, notably PVM [23], Express [21], and Linda [22] have evolved over the past few years, and have matured enough to permit their use for full-scale high-performance scientific computing. However, one shortcoming of these and other similar systems is that they are low-level emulators of parallel computers and thus do not effectively address the issue of *reasonable effort* parallelization. A variety of research efforts that do address straightforward parallelization are in progress. A substantial fraction of these are language-oriented (e.g. [24], [25]) and some (e.g. [20, 23]) are toolkit oriented. With the inclusion of compilers and run-time systems for parallel languages, most such efforts are targeted at a particular parallel architecture and thus frequently tend to exclude heterogeneous networked environments.

The EcliPSe project is an attempt to support straightforward parallelization while retaining the ability to execute in a variety of environments, especially those comprising heterogeneous processing elements. EcliPSe is a methodology and toolkit that semi-automatically "concurrentizes" sequential specifications of application programs and enables parallel execution on multifaceted hardware platforms. Originally designed for the stochastic simulation application domain, EcliPSe has been enhanced for use in other applications that possess a regular and symmetric interaction structure. While it is not, therefore, entirely general-purpose, applications that do exhibit such interaction properties are many, and EcliPSe greatly simplifies the task of parallelizing such algorithms.

In this paper, we describe the essentials of the EcliPSe system, focusing on demonstrating the use of the methodology as well as the use of the system for a large-scale computation in polymer physics. We executed this particular application on various configurations of heterogeneous systems, including hundreds of processing elements geographically distributed around the country. This is significant; to the best of our knowledge, heterogeneous network based systems have not previously been deployed [1] on large scales. Our experiences with application development and execution, as described herein, are very encouraging and significant, both qualitatively and quantitatively. From the point of view of application development, computational scientists who wish to exploit the power of parallel machines can use their sequential code in a transparent manner for the semi-automatic development of highly parallel codes for arbitrary networked machines. This frees such scientists from the burden of having to learn machine-specific parallel programming, and more than ade-

[1] Observe that this effort is different from the off-line computation efforts [26] of Lenstra and Manasse.

quately satisfies Bailey's *reasonable effort* criterion. Second, computational scientists who wish to exploit massive parallelism *interactively* can have access to a powerful environment that can be used *by tens or hundreds of others simultaneously.*

Quantitatively, our results demonstrate a high degree of effectiveness from both the performance and price-performance points of view. The application we describe is an important and inherently sequential application in polymer physics. Though the algorithms used by this application are well-known and have been used in the past with great success, this success has been limited to problems of smaller sizes. Traditional supercomputer (i.e., CRAY Y-MP) requirements for realistic problem sizes usually exceed resources available to individual researchers because scaling up the problem to a reasonable size makes the cost of executing it on a CRAY (i.e., several days to several weeks of CPU time) prohibitive.

The EcliPSe system approaches the problem by creating a virtual tree-structured machine that can accomodate virtually any computing node that can be accessed from a central monitor node. The tree-structure is built into the EcliPSe parallelization frame-work for fault-resilience, a property invaluable for long-running computations on environments capable of sudden outages. In eleven different experiments, each corresponding to a particular network configuration of arbitrarily chosen machines, we used between 16 and 192 geographically dispersed processors. Using Sparcstations, IBM RS6000s and Intel i860 nodes on distant communicating hypercubes, we show that all our virtual tree-machines outperform the CRAY Y-MP in the categories of price-performance, sheer performance, and parallelization. For example, a 127-node configuration made up of Sparc and IBM RS6000 workstations offered a computing power that was 68 times as effective as a CRAY Y-MP, where effectiveness is defined as the ratio of MIPS per Million dollars given by the CRAY versus EcliPSe. The same environment reduced a 24-hour CRAY Y-MP computation to less than 1.5 hours on the network. Finally, the transparent parallelization provided by EcliPSe outperforms the CRAY's vector compiler despite compiler optimizations.

## 2 The EcliPSe System

The EcliPSe software system was originally designed to permit straightforward and semi-automatic concurrent execution of stochastic simulation applications in a variety of parallel and distributed environments. Stochastic simulation is a technique that allows an analyst to mimic a stochastic process of interest. This application domain is inherently data-parallel in nature; furthermore, typical implementations possess well-defined and rigid control structures, enabling semi-automatic parallelization using a toolkit based approach. As the EcliPSe project evolved, it became apparent that several other classes of applications also exhibit similar characteristics, and the system has undergone enhancements to cater to such applications as well. In the simulation domain, EcliPSe takes as input a sequential model, along with user-defined constructs such as routines for pooling statistics, and transparently replicates the sequential model across processors in an arbitrary heterogeneous computing environment. The toolkit's flexibility lies in its ability to cater to a variety of simulation (or simulation-like) problems, a variety of multiprocessing schemes (e.g., replication, distribution, hybrids), and a variety of supporting architectures (e.g., shared memory, distributed memory, heterogeneous wide-area networks).

## 2.1 An Overview of the EcliPSe Toolkit

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. 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 EcliPSe system is a set of run-time libraries that perform the required simulation and control functions during the execution of the simulation. These libraries replicate sections of a simulation application, and manage communication, synchronization, and control of the multiple instances.

In using the EcliPSe toolkit, an application programmer defines the control structure of a simulation using toolkit primitives, embedded in a procedural host language. For many applications, this involves tailoring one of several predefined toolkit-provided templates, based on the "typical" structure of several classes of stochastic simulations. Even in those instances when an application does not conform to a system-provided template, the user is only required to provide a *sequential* description of the simulation, written using EcliPSe primitives. The system transparently converts this specification (or a tailored template) to a parallel program, thus hiding details regarding the existence of, and communication between, multiple processes. Both C and Fortran interfaces are available in this software library.

## 2.2 Application Development in EcliPSe

The EcliPSe system is partly based on the premise that stochastic simulation and other similar applications inherently possess a standard structure composed of actions that may be encapsulated in well defined modules:

- Data Input : Many classes of 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.

- Control Input: Inputs that influence the execution of a simulation 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.

- Sample Generation : This activity is the core of the simulation, and generates samples from the process of interest, based on the simulation. A 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.

```
. . .
/* Declarations, application dependent initializations */
. . .
setoptions(AUTO,100000,5,10,CENTRALMON,0)
/* Max samples, sample length, grainsize, monitortype */
. . .
setupdata(a,2,1000,1000,float)
/* Specify name, size and type of matrix */
. . .
/* Other setupdatas for precision, thresholds, etc */
. . .
inputdata(FUNCTION,createmat(a,1000,matrixseed))
/* Matrix to be created by user-supplied function */
. . .
/* Other inputdatas 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(random_walk(dseed,1000,a,...,totsamp,cur))
/* Sample generator - replicated by EcliPSe */
  putstat(cur)
  combine(vector_comb(totsamp,cur))
  termcheck(vector_confid(totsamp,1000,cur,...,prec,term))
/* But statistics are collected, combined and checked
only by "monitor" instance(s) */
}
if (term == 1)
  report(NORMAL1,totsamp,left,cur[0],right,cur[1])
if (exceeded(upperthreshold)) {
  simprint("Abnormal termination")
  simprint("(format string)", totsamp, cur[0], variance)
}
. . .
simterminate()
. . .
```

Figure 1: An Application Example in EcliPSe

- Statistics Combination : The samples generated in a simulation are usually combined in an application dependent manner to form an estimate.
- 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.

The EcliPSe system provides a variety of constructs in each of the above categories. The detailed syntax, semantics, and examples of these constructs are described in [20]. Figure 1 shows only a skeletal outline of the core of an EcliPSe application to illustrate the use of these constructs.

In the above example, "random_walk" is a user-supplied (sequential) sample generator which contains application specific code to generate samples. EcliPSe provides a number of library routines for random number generation, statistics combination, and even some standard sample generators that may be used in constructing this, the main simulation module. The statistics combination and confidence interval routines ("vector_comb" and "vector_confid") are toolkit-provided; if desired, a user may substitute application dependent routines. To concurrentize a simulation therefore, the user is only required to write a (sequential) sample generator, and construct a "driver" module akin to one shown in the example above – or use a system provided driver. This specification is now executable (after EcliPSe preprocessing and binding) on a variety of parallel and distributed environments. It should be noted however, that this example illustrates an application with a deliberately simple control structure; more complex structures including dynamic adjustment of the simulation parameters and using multiple sample generators are also possible and permitted in EcliPSe.

### 2.3 The Translation Phase

The translator component of the EcliPSe system is a preprocessor that parses the EcliPSe constructs in a simulation program and generates environment-dependent calls to the built-in libraries. The primary 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. Other aspects handled by the translator include variable argument lists, generation of data structure declarations where dynamic memory allocation is not possible, etc. The output of the translator is ready-to-compile source code that, when linked against machine dependent EcliPSe libraries, can execute in concurrent mode.

### 2.4 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 EcliPSe constructs as machine-dependent libraries. The significant features of the EcliPSe implementations for the different target environments are described in this section.

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 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 become 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 statistics pool is organized as a three dimensional array, with one plane corresponding to each instance of the simulation program. The `putstat` construct and the statistics combination routines use special, low-level, EcliPSe primitives to ensure synchronized access to the pool.

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 EcliPSe translator converts the control flow constructs into code sequences that send and receive messages between the multiple instances of the simulation program. Only the node processors of the hypercube are used, with node 0 assuming responsibility for input and output. 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 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.

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, 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 drawbacks. Among the disadvantages are usual limitations on the number of simultaneous connections (limiting scalability), and the overheads of initial setup. However, the EcliPSe system provides optional support software that implements a connectionless, reliable protocol to overcome these limitations.

### 2.5 An Example: Multidimensional Integration

The EcliPSe system has been successfully used to implement and execute a large number of stochastic simulation programs. Detailed descriptions of the experiments themselves and program characteristics are reported in [16]. In this section we present one representative example, with empirical measurements and speedup data. In the experiment described below, the combining method used was to require an equal number of samples from each replicated instance [6]. 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.

Prior to describing this application and our results on executing it under EcliPSe, a brief note on the interpretation of results is in order. The table shown in this section report on timings, number of samples, and speedups obtained in each of six different environments — Intel hypercubes (both iPSC/2 and RX/860), Sequent shared memory multiprocessors, a local network of IBM RS/6000 workstations, a local network of Sun SparcStations, and a wide area network of heterogeneous machines (SparcStations, RS/6000 workstations, and Sequent multiprocessors). In each case, one of the processing elements was performing *only* monitoring activities — thus, speedups are based on one less processor than shown in the table headings.

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

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

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

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

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

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

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

Table 1 shows the times in seconds, number of samples, and speedups obtained for this application using the EcliPSe system, in each of the six environments described above. This experiment, one of many we have conducted [16], demonstrates the extremely attractive speedups attainable by simple strategies for concurrent stochastic simulation. Even for applications that exhibit significant load imbalance, efficiencies of 80% and above could be achieved, thus indicating the viability, effectiveness, and efficiency of our approach.

# 3 Polymer Chains and Scale-Invariant Phenomena

The particular problem studied in this work is one in which some fundamental aspects of the statistical mechanics of polymer solutions [4] are investigated. An especially delicate and controversial problem in this area is that of the conformational properties of polymer chains in disordered media. The question is: how is the conformation of a macromolecule in

Hypercubes (**R**: Intel RX/860; **I**: iPSC/2)

| | | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| **R** | T | 770 | 275 | 118 | 56 | 27 | 13 |
| | N | 27.35 | 27.42 | 27.15 | 27.24 | 27.17 | 27.32 |
| | S | 0.96 | 2.70 | 6.31 | 13.3 | 27.6 | 57.3 |
| **I** | T | 12726 | 4252 | 1811 | 851 | 411 | 202 |
| | N | 27.3 | 27.5 | 27.54 | 27.2 | 27.32 | 27.44 |
| | S | 0.98 | 2.94 | 6.91 | 14.71 | 30.46 | 61.98 |

Other (**Q**: Sequent; **R**: Rios LAN; **S**: Sun4 LAN; **W**: Heterogeneous WAN)

| | | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| **Q** | T | 6419 | 3213 | 2141 | 1615 | 1286 | 1073 |
| | N | 27.35 | 27.44 | 27.15 | 27.32 | 27.25 | 27.32 |
| | S | 1 | 1.99 | 2.99 | 3.97 | 4.99 | 5.98 |
| **R** | T | 531 | 266 | 180 | 138 | 107 | 91 |
| | N | 27.92 | 27.35 | 27.35 | 27.42 | 27.28 | 27.35 |
| | S | 1 | 1.99 | 2.95 | 3.84 | 4.96 | 5.83 |
| **S** | T | 1492 | 783 | 512 | 392 | 346 | 292 |
| | N | 27.31 | 27.40 | 27.36 | 27.35 | 27.39 | 27.8 |
| | S | 1 | 1.96 | 2.78 | 3.78 | 4.81 | 5.88 |
| **W** | T | 1498 | 620 | 2017 | 1496 | 1149 | 962 |
| | N | 27.34 | 27.6 | 27.39 | 27.2 | 27.15 | 27.18 |

Table 1: Times and Speedups for Integral estimation

solution affected when the solution is confined to a quenched random medium ? The potentially relevant experimental realizations include polymers trapped in porous media (such as a sandstone), transport of polymer through microporous membranes, and gel permeation chromatography. Despite rather intensive theoretical and numerical studies in recent years, the problem remains unsolved, with results which seem to contradict one other.

Besides interest from the materials application point of view, there is also a more fundamental theoretical interest. This is due to the close relationship that exists between the problem of long chain polymers and that of scale-invariant phenomena at the critical point [18] (such as in liquid-gas phase transition or ferromagnet-paramagnet phase transition). This relationship has been established theoretically and demonstrated experimentally in a most convincing manner by various French groups, such as that of de Gennes [3] and that of Jannink [8]. It is the recognition of such links, based on the ideas of *scaling* and *universality*, among seemingly totally unrelated problems, that led to the recent Nobel Prize for de Gennes. In the case of polymers in disordered media, it is possible to have two different critical behaviors competing simultaneously, since the polymer size can be very large and the disorder in the medium can be long-range correlated. Such competition of two large *length scales* (in the jargon of statistical physics) makes this problem a particularly delicate and fertile ground for theoretical investigations using the tools of critical phenomena.

In this work, we focus on a linear chain which has a restricted interaction with the medium; that is, there are forbidden regions (infinite energy barrier), and the chain is confined to other parts of the medium. If the forbidden regions occur randomly and the minimum length scale of these is much smaller than the size of the polymer in homogeneous media, then this problem can be modeled, after suitable graining, by a self-avoiding walk (SAW) on a randomly diluted lattice. Thus, we first create a *percolation cluster* [19] on a finite grid (of size $L^3$) by randomly diluting the grid (i.e., removing sites as inaccessible to the chain) with probability $1-p$. The remaining sites then form connected components called *clusters*. Above a certain threshold $p_c$, there exists one cluster which spans the grid from end to end, and this is the cluster of interest. On this *disordered* cluster, a starting point of a SAW is chosen randomly, and then *all* SAWs of a predetermined number of steps $N$ are generated by a depth-first type search algorithm. At each of the $N$ steps, various conformational properties of the chain are measured, such as the moments of the end-to-end distance $R_N$ and radius of gyration $S_N$ and those of the number of chains $C_N$. These are then averaged over the ensemble of SAWs on the particular disorder configuration. This is repeated for a large number of disorder configurations, and finally both linear and logarithmic means are calculated over the disorder ensemble. Examples of chain conformations are shown in Figure 2 where, for the purpose of illustration only, a two-dimensional square grid is used and three different conformations (and starting points) of a chain of 150 steps are shown.

By virtue of the link between long-chain polymers and scale-invariant phenomena at the critical point, each of the quantities that are calculated will have a *universal* dependence on $N$ for large $N$, involving power-laws with appropriate characteristic *critical exponents*. Here *universal* means that the values of these exponents do not depend on many details of the system but only on a few fundamental specifications of the system (such as the dimensionality, and whether the chain is linear or branched). The question of main interest is whether these exponents are affected by the presence of disorder, particularly at the critical threshold $p_c$ of the dilution parameter (at which the medium has an infinitely long-ranged correlation [19]). For example, while the chains in diluted space tend to be more stretched than those in filled space, whether the exponents have different values or not is far

Figure 2: Examples of chain conformations on a critically disordered cluster are illustrated using a $150 \times 150$ square grid in two dimensions. The chains are 150 steps long and the substrate is at the critical dilution where $p_c \approx 0.593$ for the square lattice.

from obvious. Previous calculations, both analytical [5, 12, 13] and numerical [9, 11, 10], have resulted in a whole range of possible answers to this question as discussed further below. The delicate nature of this issue can be attributed in part to the wide distributions of the quantities measured, particularly at $p_c$ where the structure of the cluster is intricately *self-similar*[14].

### 3.1 Critical Behavior of a Polymer Chain in Critical Disorder

In this section, we discuss in more detail the question of the *universality class* associated with the asymptotic behavior of a SAW on a quenched disordered fractal such as our critical percolation cluster.

Historically, this question was first considered by applying an earlier general result known as the Harris criterion, which is applicable to certain statistical models of the magnetic critical phenomena in randomly diluted space, directly to the polymer problem by using the correspondence between these two problems (but in non-diluted space) introduced by de Gennes.[3] This naive application implies that the critical behavior (in particular, the critical exponents) must change as soon as any dilution is made to the space accessible to the polymer. On the other hand, there have been theoretical arguments supporting the special nature of the polymer problem which would not allow the correspondence for the non-diluted case to be carried over naively to the randomly diluted case. Furthermore, Monte Carlo simulations of Kremer[9] and some theoretical arguments indicated that the *universality* change occurs *at* $p_c$ but not for less dilution. While this latter result was supported by other calculations such as the renormalization group and exact enumeration, a different type of Monte Carlo simulation[11] seemed to indicate a very similar critical behavior even at $p_c$. More recent renormalization group study[12] contradicts all of these previous results and claims that the *universality* changes occur *twice*, i.e., at $p = 1$ and again at $p = p_c$. Most recent numerical studies appear to support this last claim. The current situation is far from clear, however, which makes this problem an important testing ground for various numerical and theoretical methods for critical phenomena in disorder, as well as this doubly *critical* problem being an important one in itself.

The troubled history of this problem suggests that there are a number of complicated features hidden in the simple model. The least subtle of these features is the type of averaging over disorder. If one takes a quenched disorder average of chains starting from a given origin (keeping in mind that disorder is defined with respect to the specific origin), then the mean square radius of gyration must be:

$$\overline{\langle R_g(N)^2 \rangle} = \frac{\sum_{\mathcal{C}} P(\mathcal{C}) \langle R_g(N)^2 \rangle_{\mathcal{C}}}{\sum_{\mathcal{C}} P(\mathcal{C})} \sim N^{2\nu}, \tag{1}$$

where a bar indicates averaging over disorder, $\mathcal{C}$ denotes a disorder configuration, and $P(\mathcal{C})$ is the probability for $\mathcal{C}$ to occur. The mean normally defines the critical exponent $\nu$ which is the analog of the correlation length exponent in a corresponding thermal critical phenomena, as described in the last line of this equation. The quantity $\langle R_g(N)^2 \rangle_{\mathcal{C}}$ is the mean square radius of gyration over one particular configuration $\mathcal{C}$. Such *quenched* averages are in general quite distinct from *annealed* type averages, e.g., the average which results by weighting every SAW of $N$ steps equally regardless of which disorder configuration the walk lives on.

Other potentially relevant questions include that of the disorder configuration ensemble. It is possible to average only over *indefinitely* large configurations (or *incipient* infinite clusters) as we do in this work, or it is also possible to average over all clusters that support an $N$ step SAW; or anywhere in between. The question of whether to take just one starting point (or chain *end* location) per configuration or many (and if so, with what *weight* given to individual starting point) also presents itself. The latter question is relevant to possible experimental realizations since in any experiment measurements will reflect some average over the positions of the polymer in the medium. This is also an important issue for simulation if one is to compare the results of a simulation based on a chain that is relaxed by moving them through a disordered medium with those of a simulation based on generating SAWs from a fixed origin (as we do in this work).

There are also other more subtle features. One is the question of the difference between *walk* statistics and *chain* statistics. In this context, a *walk* has a statistical weight determined by the product of $1/z$ at each step, where $z$ is the local coordination number (or the number of ways the next step can be taken avoiding forbidden sites due to dilution). In contrast, a *chain* has an equal statistical weight with any other chain of the same length.

Another hidden feature that is important is the broadness of some of the statistical distributions associated with this problem. Consider, for example, the number of chain conformations of $N$ steps starting from a fixed origin on a diluted cluster. The mean of this quantity, $\overline{C_N}$, normally defines another critical exponent $\gamma$, which is the analog of the susceptibility critical exponent in corresponding thermal critical phenomena:

$$\overline{C_N} = \frac{\sum_{\mathcal{C}} P(\mathcal{C}) C_N(\mathcal{C})}{\sum_{\mathcal{C}} P(\mathcal{C})} \sim N^{\gamma-1}\mu^N \tag{2}$$

Since $C_N$ is a multiplicative random variable, its distribution can be extremely broad. In fact, our previous enumeration study[10] showed clearly that the most probable value and the true mean are far separated in this case, and suggested several interesting possibilities. In experimental realizations, one or both ends of the chain can be frequently *trapped* in the region of high connectivity and measurements may reflect most probable values of a variable (and not the mean) over the disorder ensemble, and this certainly applies to (incomplete) simulations also.

The broadness of certain distributions may affect the observations even within a single disorder configuration (with a fixed starting point). The complex topological properties of critically diluted clusters may cause sufficient broadness for some moments of the size distribution so that, for example, measured radius of gyration may reflect averages over the free end of the chain *trapped* in some subset of the cluster, possibly in any anomalously high connectivity region (called *blobs*), most of which are within the *dangling ends* even though an indefinitely long chain can only exist on the so-called *backbone* (or biconnected) part of the cluster.

An example of a previous enumeration result[15] is shown in Figure 3 where the estimation of the exponent $\nu$ for the *chain* statistics (see above) is compared against that of the corresponding exponent for the *walk* statistics. The data shown are for the incipient infinite cluster only, and only one starting point was used per disorder realization. The estimation works by first computing an *effective* exponent $\nu_N$ which makes use of data only up to $N$ steps, and then the true $\nu$ is obtained by extrapolating $\nu_N$ as $N \to \infty$. The analysis of these data seems to suggest that the two different statistics may actually result in essentially the same estimate of $\nu$ which is about $0.65 \pm 0.01$ (but significantly distinct from the undiluted case of about 0.59). The fact that the exponent $\nu$ increases at $p_c$ compared with $p = 1$ is in agreement with the expected universality change; however, the apparent coincidence of the estimates for the two different ensembles is surprising since the chain statistics are expected

Figure 3: We show the effective exponent $\nu_N$ (which uses data up to $N$ steps) for the radius of gyration, $R_g$, and the root-mean-square end-to-end distance, $R_e$, of the self-avoiding chains and walks on the simple cubic lattice at $p_c \approx 0.312$. This figure is taken from Moon and Nakanishi.[15]

to enhance the highly connected regions much more than the walk statistics. In fact, for the square lattice in two dimensions, the corresponding results do indicate a significant increase of $\nu$ only for the chain statistics and *not* for the walk statistics.

This is just one of the many puzzling points that arose from the often contradictory numerical calculations performed in the past. To better understand these different effects, we really need to extend the exact enumeration calculations which have the unique advantage of using the complete ensemble rather than sampling. This eliminates potential biasing and allows for the calculations of the entire distributions. However, the obvious disadvantage of the method is that it is limited to relatively short chains, making it nearly impossible to probe effects that become important only for very long chains. Note that because of the limitations of the capabilities of conventional computing platforms (in this case, a Kubota Pacific Titan and an ETA10* at Purdue), the largest number of steps $N$ attainable previously was 35, and the number of independent disorder realizations was about 1500 (on the simple cubic lattice) in the case of the calculation shown in this figure. The objective of the current work, on the physics side, is to extend the usefulness of the enumeration method by finding ways to significantly increase both the number of steps $N$ and that of the disorder realizations, by taking advantage of massively parallel computing utilizing widely scattered idle resources over long-haul networks.

### 3.2 Sequential Nature of Underlying Algorithms

Through the following description, it will become readily apparent that this class of polymer based random-walk problems in three dimensions are indeed a challenge for fast scalar or vector processors. Through a detailed study of the code, we determined that the algorithms exhibit a performance hampering behavior due to a high non-locality of data referencing caused by the cluster generation and search phases in the large random structures (approximately 700,000 to 1 million integers, and 15,000 double precision numbers). In addition, both the clustering algorithm and the depth-first search algorithms described below are inherently sequential, and primarily utilize integer arithmetic for cluster generation and SAW enumeration. The two major subroutines used in the sequential algorithm are *make3* (which generates clusters and checks for connectivity) and *wsaw* (which performs a compute-intensive depth-first search). Both the routines used are based on well-known algorithms that have been used for several years with great success.

First, subroutine *make3* uses a Monte Carlo scheme to generate a percolation cluster that is a maximal connected component of occupied sites. This cluster must span the cubic grid in all three directions simultaneously. This is done using an algorithm due to Hoshen and Kopelman [7]. Thus, we decide whether each site of the grid is occupied or not by comparing a $U(0,1)$ (i.e., uniformly distributed between 0 and 1) random number to an input parameter $p$ which represents the fill fraction. This is done one at at time, column by column, and layer by layer. The nontrivial part of the algorithm lies in maintaining connectivity information. Initially, an occupied site $(i,j,k)$ is labeled by a cluster number $latt(i,j,k)$ (which labels each connected component). However, when a newly occupied site joins two or more pre-existing clusters, a problem arises. While a simple scheme suggests relabelling all sites by a common cluster number, this is prohibitively time-consuming. Thus instead of relabelling sites that were already labeled, we keep a separate *label tree* which records information about which clusters are in fact connected to which other clusters. In this way, the *tree* keeps the pointers (using a value that is the negative of the cluster number it points to) *and* also the size of the cluster, if the index is the true or the minimum cluster label. That is, if $label(i) = -j$ (with $j > 0$), then the true cluster label of the cluster $i$ is $j$; if $label(i) = j$ (with $j > 0$), then $i$ is the true label and the cluster size is $j$.

After the cluster generation, the maximal cluster is identified, and then checked to determine if it spans the cube in all three directions. If so, the periodic boundary conditions are applied in all directions. Finally, a *link* matrix is created from the lattice map of the percolation cluster. This array is two-dimensional with the first index being the site number and the second the number indicating the nearest neighbor. The value of $link(i, j)$ is the site number of the $j$-th neighbor of the $i$-th site of the cluster. If a particular neighbor is empty, then the corresponding entry is given a value equal to its own site number. This information is used in the next step when self-avoiding walks (SAWs) are generated on the cluster.

Next, we discuss the subroutine *wsaw* which enumerates all SAWs of *nstep* steps starting from a fixed starting point (site *init*) by a depth-first type search algorithm. Thus, each time a step is added, we always try to add an occupied nearest neighbor that is the first unattempted neighbor in the order listed in the array *ndr*. The array *link* created by *make3* tells whether the particular neighbor is occupied or not and if so what the site number of the neighbor is. If this is a success, i.e., if it has not been visited by a previous step, ($mocc(nsite) = 0$), then it is accepted into the ensemble and various properties are measured, and their moments updated. After that, the next step is immediately attempted to extend the length of the SAW. On the other hand, if the attempt fails (because that neighbor has been visited by the same SAW before), then this branch of the SAW is finished. Thus, the SAW must retract one step, and begin a new branch in the depth-first search with the same preference order of the search directions. When the search is retracted all the way to the starting point, the whole *tree* is complete, and control returns to the calling program.

## 4 Parallelization with EcliPSe

There is little evidence to suggest that the best way to parallelize a computation for which code already exists is to spend costly man-hours in redesigning and recoding the computation for a particular parallel machine. While success stories for such efforts can be found in the literature, we know of few computational scientists who are willing to devote their careers to parallelizing existing codes. On the other hand we have been associated with several computational physicists, chemists and biologists who cannot afford to devote time and effort to issues of parallel programming, but yet have existing codes or problems that simply cannot be tackled in a reasonable way without the power of massively parallel processing. The EcliPSe software prototype was motivated by our recognition of precisely this problem.

The polymer simulation problem is typical of most Monte Carlo problems in that it possesses a simple repetitive structure. The main routine initializes various arrays for statistics and calls a slave routine which goes about the business of sampling. In this case each sample corresponds to a vector (i.e., $28 \times nstep + 12$ double precision numbers) of statistics computed via the depth-first search, after the random generation of clusters. The slave routine computes statistics for each new sample obtained, as well as cumulative statistics. Either through use of appropriately chosen seeds, or through use of a parallel random number generator, such simulation applications lend themselves to massive parallelism in a variety of ways, each of which is fully exploited by EcliPSe. In some case, such as in particle simulations in molecular dynamics, EcliPSe distributes the computation across sets of processors and then replicates these configurations across all available processors, balancing problem distribution between processors with replication across processors. In the polymer simulation problem discussed here, the sequential nature of the clustering and SAW generation problems precluded efficient problem distribution, and we focused instead on replication. A few hours of effort were required in parallelizing the original code using one of a variety of EcliPSe templates.

In addition to the template based parallelization, EcliPSe requires that the user develop a statistics combination routine for real-time pooling together of statistics computed by the different sampling processors. Results are displayed and interpreted on-line at the simulation-initiating processor as soon as they are received and combined by this processor. It should be noted that files and I/O are *not* used in doing statistics combination; these combinations are interspersed with sample computations, with a frequency controlled through a user-input granularity parameter. Several combining strategies are provided by the toolkit, along with some facilities for constructing unbiased estimates. It is particularly useful to pay attention to obtaining proper estimates because the added dimensionality brought about by parallel sampling in simulation can cause significant bias [6] through order-statistic related problems.

### 4.1 On EcliPSing Polymer Computations

To demonstrate the ease with which a production run can be made, starting with a sequential program, we give a brief description of the process of parallelizing the polymer code. As explained in section 2.2, the code consists of a set of declarations, and three computational units: spanning-cluster generation, depth-first-search, and statistical calculations. The last computational phase requires less computation than the cluster generation and SAW enumeration or depth-first search phases.

Figure 4 shows the general flow of control for all replications of the parallelized code. Declarations and initializations made in the original code remain intact in the parallelized version. The *setlc*(*hosts*) block is the first of three EcliPSe blocks that are selectively executed only by the monitor. Here, the host configuration file is read in and the set of available processors defined. The *setoptions* command ensures that all processors will be used (AUTO) to set up a tree architecture (TREE). The parameters *nsamp* and *len* define the number of samples to be reported by each processor before termination, and the size of each sample. The fourth parameter defines the granularity of sampling (i.e., every sample is returned, with accumulation) and the last parameter defines the level of EcliPSe primitives used.

The *inputdata*() and *inputseeds*() commands read in and transparently communicate problem inputs and random number seeds to all the processors in the tree. This is done through an efficient process of data-diffusion down the tree, from the root to the leaves. The number of steps required for this is equal to the height of the tree. Control then moves to an original program block, responsible for performing a set of input dependent initializations. At this point the stage is set for the primary computations: percolation cluster generation, walk generations, and statistical ensemble averaging.

Figure 5 shows the flow of control in the compute-intensive portion of the computation, within each sampler. After further initializations, each processor generates a spanning cluster on a three-dimensional lattice. An infinite cluster connecting the $x-$, $y-$ and $z-$

directional boundaries is determined using the Hoshen-Kopelman algorithm. Once such an infinite cluster is generated, periodic boundaries are used to connect the otherwise disconnected edges. When this procedure is complete, a depth-first search is initiated for enumeration of all self-avoiding walks of a given length. Both during and upon completion of the depth-first search, various conformational properties are computed. For each disorder configuration, ensemble averages of SAWs are computed and accumulated, and this is repeated for many disorder configurations. One-batch averages, linear one-batch averages, and logarithmic one-batch averages are computed and accumulated. All these computations take place within procedure *compute_I*(). The granularity parameter in EcliPSe may be used to control the batch size in each accumulation.

When accumulated statistics are ready, the *putstat*() function is invoked for sending data up the tree. A vector of statistics is passed up the virtual tree machine, from each node to its parent in the tree. Each parent node combines the data sent to it by its children with its own data and passes this data along to its own parent. This "data-combining" process achieves a tremendous reduction in the amount of data sent up the tree to the monitor process, effectively preventing the merging of samples from causing a bottleneck at the monitor.

As can be seen in Figure 4, the process of computing and merging continues until statistics are obtained from a predefined number of batches, or a given degree of statistical accuracy had been achieved. A simple check determines whether each processor must proceed with the computation or stop. Observe that each node must interpret the *putstat*() function according to its position in the tree: leaves only send data, while parents must combine and send. When the termination condition is finally satisfied, the monitor computes overall statistics in procedure *compute_O*(), displays results, gracefully shuts down the operation of the distributed computation and terminates.

### 4.2 Parameters and Results

The parameters for the work presented here are in the range of having maximum impact on the resolution of the scale controversy described in Section 3. The value of $p = 0.312$ represents a good numerical approximation to the critical threshold for the percolation problem on the simple cubic lattice. Also, while as large a value of $N$ (number of steps) as possible is desired, the computational time increases exponentially with $N$. Since previous Monte Carlo simulations [11] have indicated a mysterious crossover of the $N$ dependence at about $N = 25$ to 30, the value of $N = 35$ was chosen to include this region of possible crossover. The present enumeration approach [10] then has the potential of determining whether the Monte Carlo observations reflect *entropy trapping*, i.e., biased sample space. The size of the grid $L = 35$ (with periodic boundary conditions) is chosen so that the radius of the polymer ($R_N$ or $S_N$) is much smaller than $L$ to minimize the finite size effects, and yet $L$ is small enough so that the critical percolation cluster spanning the grid in all directions can be created easily. Using 2000 clusters is roughly equivalent to what is being done in Monte Carlo simulations, and again, while as large a sample space of disorder as possible is desired, this number is comparable to the largest calculation of this type performed previously [15].

Our initial results have been surprising. We have begun conducting routine production runs on a large scale for increasing values of $N$, in order to examine the mysterious crossover of dependence on $N$ and simultaneously determine if entropy-trapping phenomena occur.

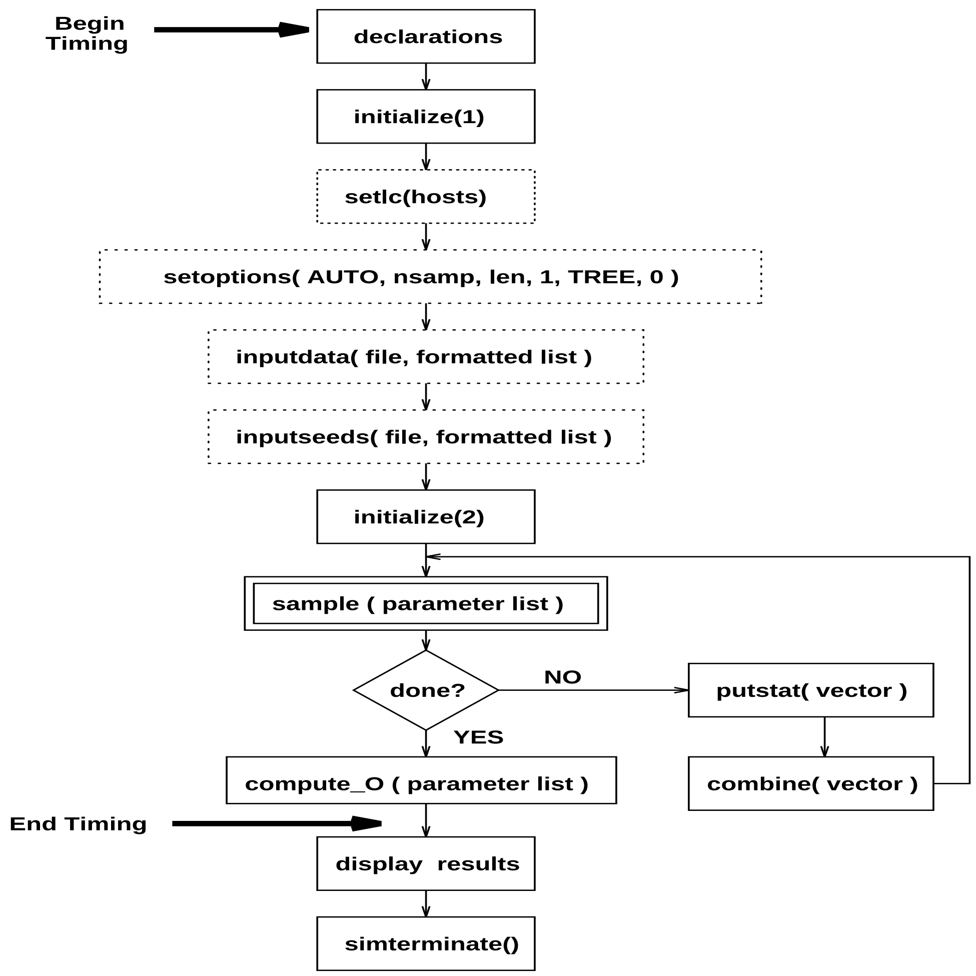


Figure 4: Main program template

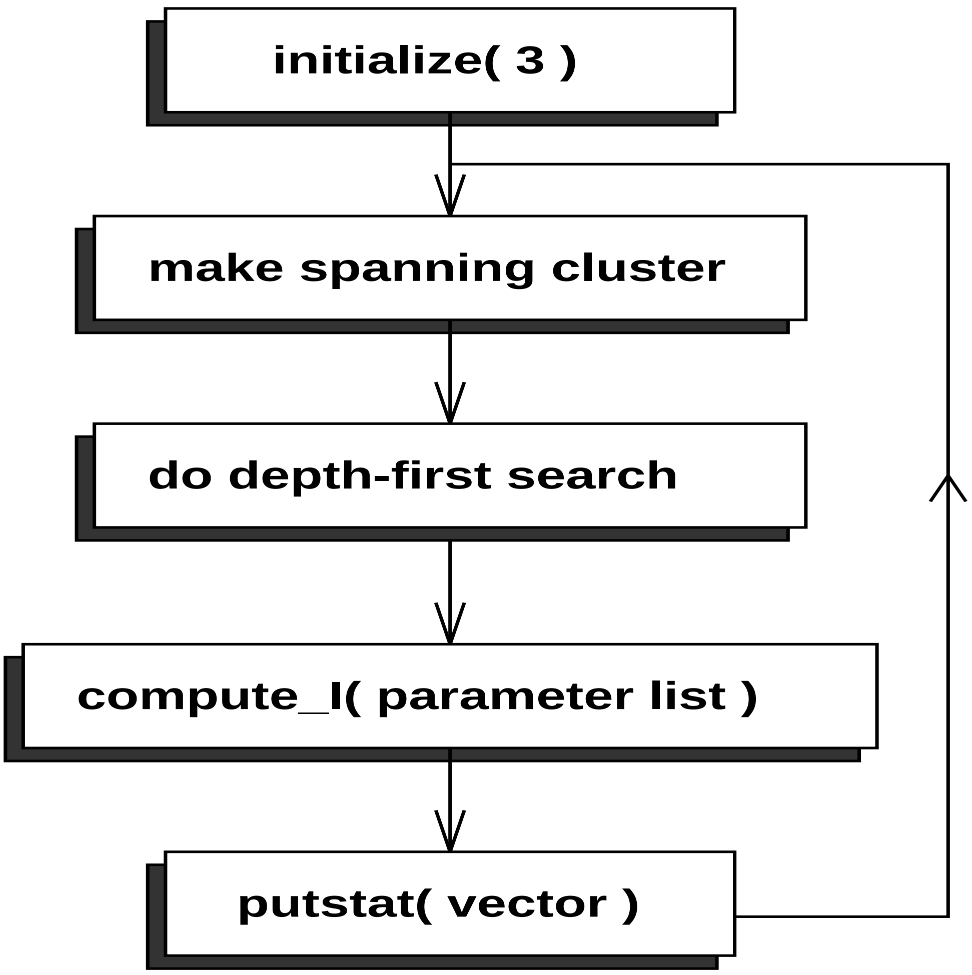


Figure 5: Sampler program structure

From a simulation viewpoint, the subject of *sampling bias* due to the complexity of the sample space is of great interest. In the absence of such a bias, there is considerable potential for improving execution times many-fold by replacing the time-consuming depth-first type search algorithm by appropriate standard sampling or importance sampling algorithms. Due to the very delicate and controversial nature of the problem, we have decided to refrain from making conclusions based on partial results. Instead, we focus primarily on execution performance in this report. Having found that the parallel environment has reduced the experimentation effort by more than an order of magnitude, we have increased the scope of our experiments.

The parallelization has made an otherwise impossible interactive experimentation process possible. Previous exercises conducted on the CRAY Y-MP were only permissible in batch mode, because of the computing time and cost involved, and allowed for little interactive experimentation. Further, limited CRAY access time made the entire experimentation process difficult. This phenomenon is also true of multiprocessors like the Intel i860 hypercube and the 512-node Delta at Caltech. Even assuming that one has obtained computing time on such machines, it is frequently difficult to allocate a subcube on the hypercube or a mesh on the Delta for an interactive simulation experiment because of the high user-competition for nodes. In contrast, our experience is that a computing environment flexibly allowing for computation nodes that are SUN/IBM workstations, Intel i860 nodes, IPSC/2 nodes or Sequent processors improves the entire interactive experimentation process by an order of magnitude. If a subset of nodes fail, or a subnetwork goes down, the rest of the computation remains active, resulting in increased computation time but a graceful degradation of service.

# 5 The Computing Configurations

For this experiment, we ran the parallelized version of the polymer chain simulation on *eleven different heterogeneous computing environments* dispersed across the country. The various machines used to build these network configurations are shown in Figure 6.

The simulation was done in parallel across the machines named in each configuration, with results being combined at monitoring nodes in the tree-structured virtual machine. The configurations we used are:

- **Configuration 1: Sixteen nodes**
  16 SUN Sparc workstations (Purdue)
- **Configuration 2: Sixteen nodes**
  16 IBM RS6000 workstations (Florida State University)
- **Configuration 3: Sixteen nodes**
  16 Intel i860 nodes (Oak Ridge National Labs – ORNL)
- **Configuration 4: Thirty-two nodes**
  32 SUN Sparc workstations (Emory University)
- **Configuration 5: Thirty-two nodes**
  32 IBM RS6000 workstations (28-Florida + 4-Emory)

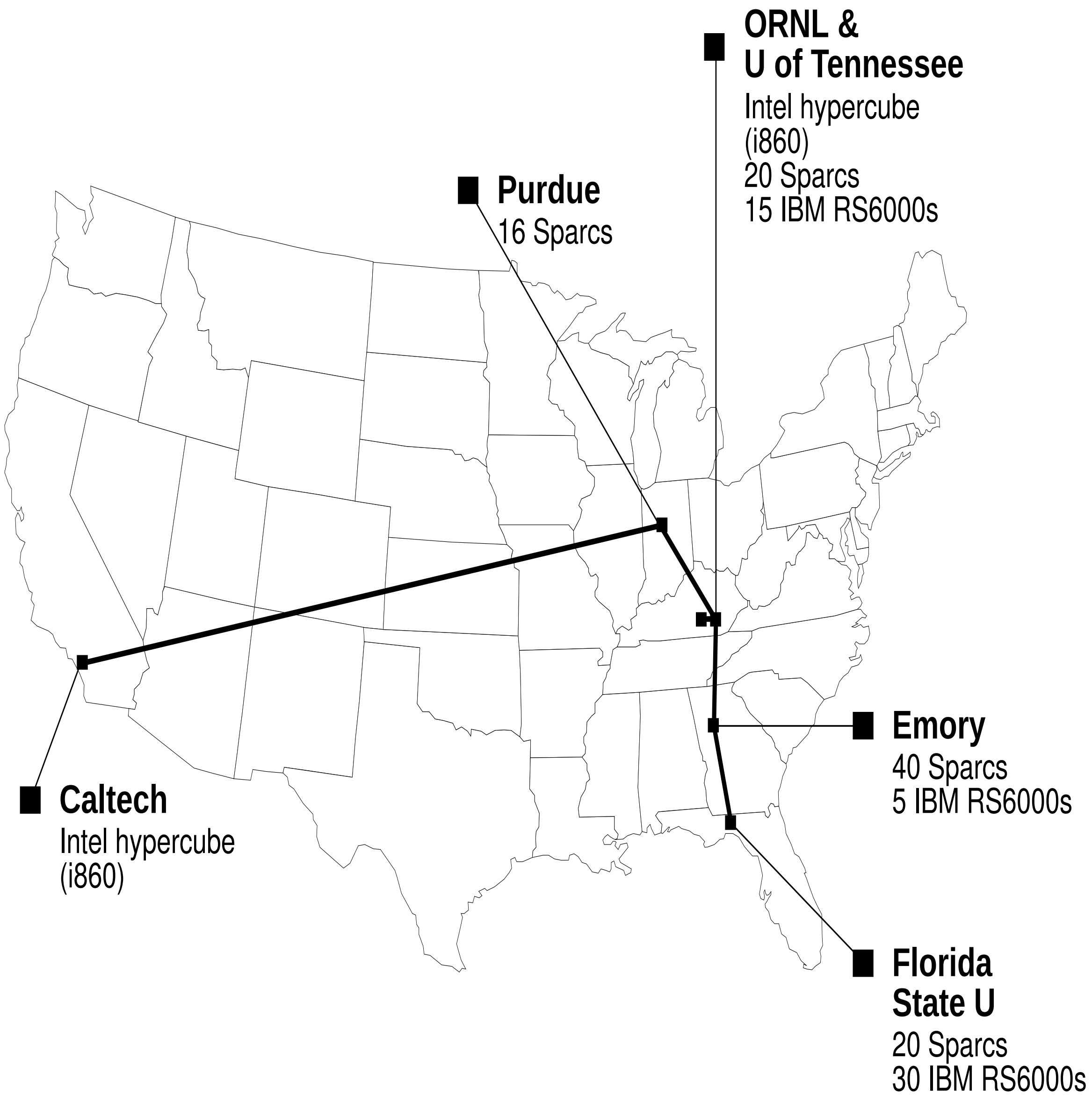


Figure 6: The Configuration Space

- **Configuration 6: Thirty-two nodes**
  32 Intel i860 nodes (ORNL)

- **Configuration 7: Sixty-four nodes**
  32 Sparcs + 32 RS6000 (32-Emory Sparcs + 28-Florida RS6000 + 4-Emory RS6000)

- **Configuration 8: Sixty-four nodes**
  64 Intel i860 nodes (ORNL)

- **Configuration 9: One hundred and twenty-eight nodes**
  48 RS6000 + 80 Sparcs (28-Florida RS6000 + 5-Emory RS6000 + 15-ORNL RS6000 + 20-Florida Sparcs + 40-Emory Sparcs + 20-ORNL Sparcs)

- **Configuration 10: One hundred and twenty-eight nodes**
  128 Intel i860 nodes (64-ORNL i860 nodes + 64-Caltech i860 nodes)

- **Configuration 11: One hundred and ninety-two nodes**
  48 RS6000 + 80 Sparcs + 64 Intel i860 nodes (28-Florida RS6000 + 5-Emory RS6000 + 15-ORNL RS6000 + 20-Florida Sparcs + 40-Emory Sparcs + 20-ORNL Sparcs + 64-ORNL i860 nodes)

### 5.1 Fault tolerant tree-structured computations

Most nontrivial stochastic simulations, even massively parallel ones, tend to be long-running compute-intensive applications. For simulations that execute for days, or even weeks, the ongoing parallel computation must be protected against nodal faults or subnetwork outages. We achieve this by arranging processor configurations in tree-structures, where a processor is a node in the tree. Computation and control information is piped bi-directionally along the branches.

Two examples of fault-tolerant tree-machines are shown in Figure 7. Recall that the nodes are arbitrary, geographically distributed machines. A simple tree construction algorithm takes as input the processor configuration file and sets up the tree structure using prescribed limits on the number of children a node can have. In Figure 7, $M$ represents a statistical *monitor*, and $S$ represents a *sampler*. The root node is the central monitor at which computation is initiated. Each sampling processor computes along an independent trajectory and reports results to its parent. Each parent node performs sampling as well as monitoring, combining its children's results with its own, before passing the combined results to its own parent in the tree. Stream connections between the machines maintain flow-control. In all of the experiments reported here, the central monitor did not perform sampling. Hence, in reporting results based on a configuration with $n$ processors, we use the number $(n-1)$ in the table to describe the number of processors involved in the computation (e.g, see the first column in Table 2).

In the tree at the top of Figure 7, the rectangles represent processors that are both monitors as well as samplers. Only the leaves are strictly samplers. If a leaf processor is to fail, it is simply ignored by the parent monitor. However, if an interior node is to fail, its parent takes over the responsibility of monitoring the computations of children of the failed node. We use a time-out mechanism to effect this, with a simple reconfiguration protocol

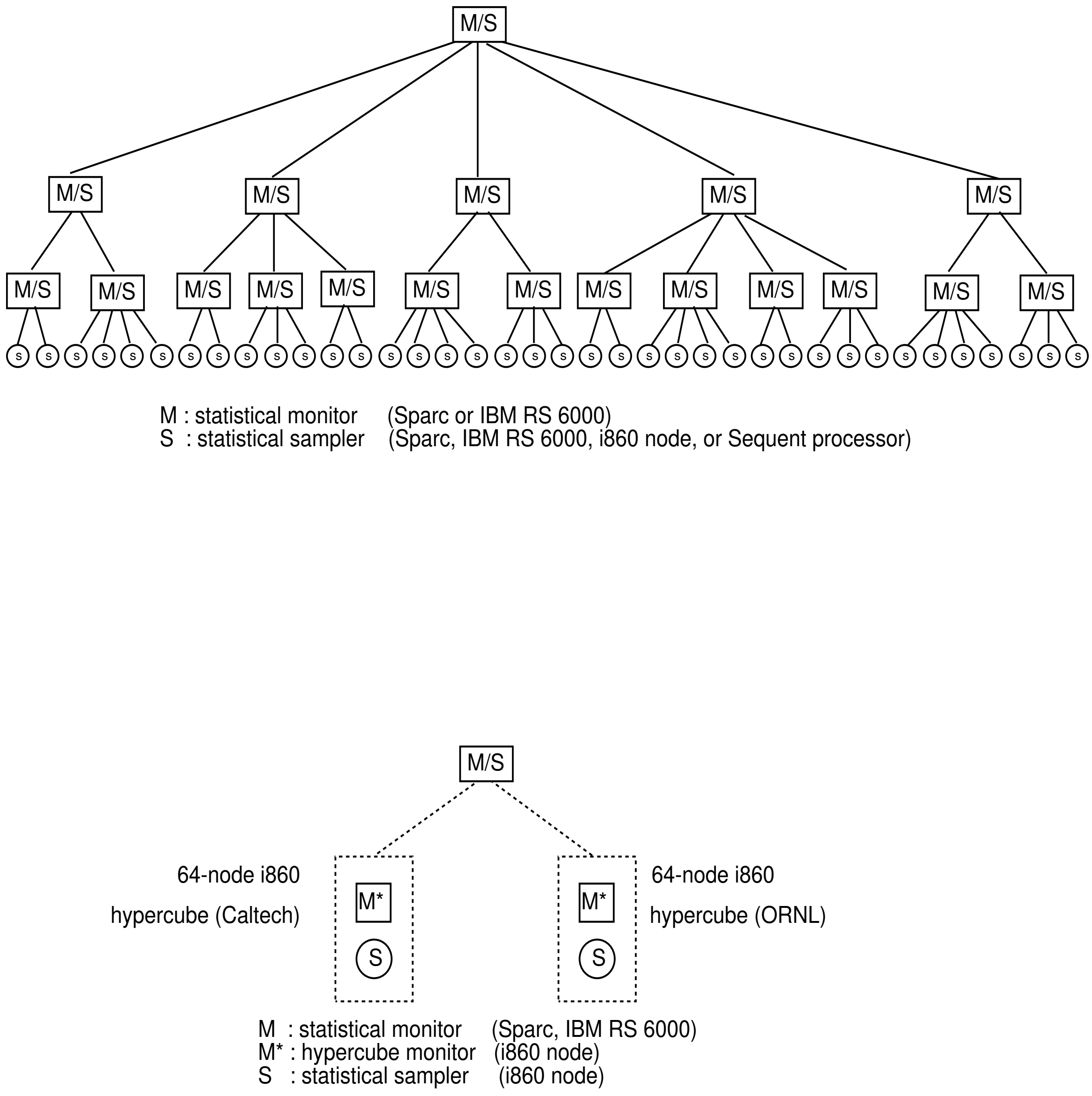


Figure 7: Tree-structured virtual machines

which appears to work well. In the worst case, if all interior nodes fail, the central monitor takes over monitoring of the entire computation. The nodes in this figure may be arbitrary processors (though we only used Sparcs, IBM RS6000s, and i860 nodes).

In the tree at the bottom of Figure 7, we used an IBM RS6000 as the central monitor, with two 64-node i860 hypercubes as subtrees, one at Caltech and one at Oak Ridge (Configuration #10); both hypercubes sampled simultaneously and reported results to monitors. Each of the subcubes devotes one node to the monitoring function. We refrained from building fault-tolerance within subcubes. It has been our experience that if a single i860 node goes down, the entire subcube fails. In our tree, this would affect one-half of the computation, but the simulation would eventually complete, requiring twice as much time as it would have required had the subcube failure not occurred.

### 5.2 Conducting the Experiments

The logistics of actually executing EcliPSe applications on a large number of geographically dispersed machines is made straightforward by built-in system facilities. The procedure consists of creating a text file containing the Internet names of the machines, as well as information pertaining to login details, location of executables, and where appropriate, the number of PE's available on the machine (e.g. the hypercube). Application programs, both samplers and monitors, have to be compiled for each architecture in the configuration, and distributed to each machine in the host pool. The executable copy of the compiled and linked object is then started on the "local" machine, which then initiates, using existing facilities, the remote copies of the monitor and/or sampler. Normally, the copy of the EcliPSe application executing on the local machine comprises the root of the tree structure referred to earlier.

As mentioned, these experiments were conducted at five geographically dispersed installations at which we had access to computing facilities. Especially in the larger configurations that we used, the use of resources on any one machine was very small (of the order of a few CPU minutes); thus our experiments could be carried out with minimal disruption to workstation owners or other users. In situations where smaller host pools were used, permission was obtained from system administrators in order to avoid infringing on other computations, but in many cases this turned out to be unnecessary, if our experiments were conducted during off-peak hours. Interestingly, in all these experiments, few difficulties were encountered owing to the physical and geographical distribution of our hardware platform. On some occasions, machines and network links did fail, but the application ran to completion, owing to the simple but effective built-in failure resilience mechanism[2]. Although formal records were not maintained, our observation was that single machine or single network-link failures occurred, on the average, at a frequency of less that one every twenty four hours. This was based on our experience with the pool of about a hundred machines, over the several-week period in which we conducted experiments.

[2]It should be noted that performance results are shown only for those experiments that ran to completion without failures.

| Nprocs | Sun | RS6000 | Sun+RS6000 | i860 | Sun+RS6000+i860 | Average | Equiv CRAY |
|---|---|---|---|---|---|---|---|
| 15 | 7101.54 | 1992.87 | - | 9226.09 | - | 6106.83 | 12735.5 |
| 31 | 3194.36 | 1366.06 | - | 4661.96 | - | 3074.12 | 13250.6 |
| 63 | - | - | 1015.47 | 2108.07 | - | 1516.77 | 11552.5 |
| 127 | - | - | 492.21 | 893.55 | - | 692.88 | 10526.4 |
| 191 | - | - | - | - | 574.06 | 574.06 | 11493.5 |

Table 2: Timing comparisons of EcliPSe vs. CRAY Y-MP (seconds)

# 6 Performance results

We performed exactly one simulation experiment in each of the eleven computing configurations described. Each of these runs used the same input parameters. As mentioned in Section 5.1, the number entered in the tables for the "number of processors used" excludes the central monitor, and hence is always one less than a power of two. To avoid order-statistic related bias [6], the central monitor requested an equal number of batches (sample vectors) from each processor. The total number of batches obtained was an integer multiple of the number of samplers used. Specifically, in runs using 15, 31, 63, 127, and 191 processors, 495, 496, 441, 381, and 384 batches were computed respectively. Thus, for example, in the 15-processor experiments, each processor computed 33 batches, using a single pair of random number seeds to initiate the entire trajectory. Similarly, 7 batches per processor (each using a single pair of seeds) were computed in all 63-processor experiments.

The yardstick against which the results are compared are *equivalent timings on a single processor* of the CRAY Y-MP system at Illinois. For each of the eleven configuration-dependent simulation experiments we conducted, we made an identical run of optimized code on the CRAY Y-MP. Thus for example, to get a CRAY run equivalent to the 191 processor run of Configuration #11, the CRAY was required to utilize the 191 distinct pairs of seeds used by the 191 processors of this configuration. The results of the CRAY run were identical to the results given by the processors of Configuration #11. These equivalent CRAY runs were made for each of the eleven experiments performed so that we could make a detailed comparison against the CRAY.

Table 2 shows the execution times, in seconds, for each of the eleven experimental runs that were conducted. Once again, we point out that each row of the table (i.e. each experiment with different numbers of processors) is slightly different from the others, owing to the different total number of batches computed by each processor configuration of a given size. In other words, the workload for each row is different, as explained at the beginning of this section.

In Table 2, the execution time, in seconds, is shown for each virtual machine configuration. For example, a 496-batch (i.e., 31 processors, each sampling 16 batches) run took 3194.36 seconds on a network of Sparcstations, 1366.06 seconds on a network of IBM RS6000

| Configuration | Composition | Cost |
|---|---|---|
| 15 Sparc | 16 SLC Sparcstations | $79920 |
| 15 RS6000 | 16 RS6000/320 Systems | $113120 |
| 15 i860 | 16 nodes of Intel i860 | $375000 |
| 31 Sparc | 16 SLC Sparcstations +<br>16 SS1 Sparcstations | $175840 |
| 31 RS6000 | 32 RS6000/320 Systems | $226240 |
| 31 i860 | 32 Nodes of Intel i860 | $750000 |
| 63 Sparc +<br>RS6000 | 16 SLC Sparcstations +<br>16 SS1 Sparcstations +<br>32 RS6000/320 Systems | $402080 |
| 63 i860 | 64 nodes of Intel i860 | $1500000 |
| 127 Sparc +<br>RS6000 | 36 SLC Sparcstations +<br>45 SS1 Sparcstations +<br>38 RS6000/320 Systems +<br>9 RS6000/550 Systems | $1042255 |
| 127 i860 | 128 nodes of Intel i860 | $3000000 |
| 191 All | 36 SLC Sparcstations +<br>45 SS1 Sparcstations +<br>38 RS6000/320 Systems +<br>9 RS6000/550 Systems +<br>128 nodes of Intel i860 | $4042255 |

Table 3: Configurations of virtual tree-machines

| Configuration (Experiment) | CRAY MIPS | EcliPSe MIPS | CRAY MIPS/$M | EcliPSe MIPS/$M | EcliPSe/CRAY Effectiveness Ratio |
|---|---|---|---|---|---|
| 15 Sparc | 44.75 | 86.72 | 17.90 | 1085.08 | 61 |
| 15 RS6000 | 44.75 | 356.02 | 17.90 | 3147.27 | 176 |
| 15 i860 | 44.75 | 61.76 | 17.90 | 164.69 | 9 |
| 31 Sparc | 44.91 | 204.14 | 17.96 | 1160.94 | 65 |
| 31 RS6000 | 44.91 | 544.10 | 17.96 | 2404.96 | 134 |
| 31 RX/i860 | 44.91 | 127.66 | 17.96 | 170.21 | 9 |
| 63 Sparc + RS6000 | 45.16 | 202.69 | 18.06 | 504.10 | 28 |
| 63 i860 | 45.16 | 126.82 | 18.06 | 84.54 | 5 |
| 127 Sparc + RS6000 | 44.62 | 1255.56 | 17.84 | 1204.65 | 68 |
| 127 i860 | 44.62 | 527.17 | 17.84 | 175.72 | 10 |
| 191 All | 44.79 | 1088.07 | 17.91 | 269.17 | 15 |

Table 4: EcliPSe MIPS vs. CRAY MIPS

workstations, and 4661.96 seconds on 31 nodes of an Intel i860 hypercube, resulting in an average time of 3074.12 seconds across all EcliPSe runs. By contrast, exactly the same results are obtained on the CRAY Y-MP by running the simulation for 31 pairs of seeds in sequential fashion, but requiring a total time of 13250.6 seconds. Similarly, a 384-batch run under EcliPSe using 191 processors (a combination of Sparcstations, IBM RS6000's and a 64-node hypercube) completed in 574.06 seconds, compared to 11493.5 seconds on the CRAY (191 sequential runs, 2 batches each) as shown in the last row of the table. As indicated in Section 2.1, the 8K byte data-cache of the i860 node increased the Intel timings because of severe nonlocal referencing, mainly in the depth-first search phase. The 64K byte cache allowed the Sparc to perform impressively.

This table illustrates the tremendous gains in overall execution times that are obtainable by utilizing a geographically dispersed collection of systems, including both general purpose workstations and hardware multiprocessors. A run requiring 4 hours on a CRAY processor could be completed in a roughly twelve minutes on a network-based massively parallel system. Furthermore, it should be noted that the workstations that were used in these experiments *were not dedicated to our runs; rather, computing resources from these workstations were "scavenged" when they were not in use for other purposes.* This, in effect, is tantamount to constructive use of otherwise wasted cycles, and therefore, demonstrates an important (albeit subjective, and perhaps moot) point. Nevertheless, it is a constructive solution to the "interactive supercomputer" issue raised by Bailey [1].

The raw performance numbers for the experiments are reported in the following discussion. However, as explained earlier in Section 3.2, the problem is an inherently poor

| Configuration (Experiment) | CRAY MFLOPS | EcliPSe MFLOPS | CRAY MFLOPS/$M | EcliPSe MFLOPS/$M | EcliPSe/CRAY Effectiveness Ratio |
|---|---|---|---|---|---|
| 15 Sparc | 5.74 | 11.12 | 2.29 | 139.13 | 62 |
| 15 RS6000 | 5.74 | 45.67 | 2.29 | 403.73 | 180 |
| 15 i860 | 5.74 | 7.92 | 2.29 | 21.12 | 9 |
| 31 Sparc | 5.76 | 26.19 | 2.30 | 148.94 | 64 |
| 31 RS6000 | 5.76 | 69.80 | 2.30 | 308.52 | 133 |
| 31 RX/i860 | 5.76 | 16.37 | 2.30 | 21.82 | 9 |
| 63 Sparc + RS6000 | 5.79 | 26.00 | 2.31 | 64.66 | 28 |
| 63 i860 | 5.79 | 16.27 | 2.31 | 10.84 | 4 |
| 127 Sparc + RS6000 | 5.72 | 161.08 | 2.28 | 154.54 | 69 |
| 127 i860 | 5.72 | 67.63 | 2.28 | 22.54 | 10 |
| 191 All | 5.74 | 139.60 | 2.29 | 34.53 | 15 |

Table 5: EcliPSe MFLOPS vs. CRAY MFLOPS

candidate for parallelization – as a result, spectacular MFLOP ratings are not observed. What is indeed remarkable are the factors by which the EcliPSe experiments outperform traditional supercomputers like the CRAY Y-MP even in terms of raw performance obtainable. In this section, we describe the hardware platforms on which the EcliPSe experiments were conducted, and lay the foundation for the most significant of our results – excellent price-performance ratios.

Table 3 shows the virtual-machine configurations that were used in each of the experiments, with details of the exact models used. The first column shows the number and type of processors; each group of "15", "31", "63", "127", and "191" configurations corresponds to one set of run parameters. The second column gives the exact configuration of the virtual machine, indicating by model, the composition of each of the clusters. Finally, the third column gives total list prices of these configurations, obtained from vendor announcements. Individual machine prices are based on minimum configurations (memory, disk etc.) required to actually execute this application.

Two points are worthy of note, with regard to Table 3. First, the specific machines in each composition are those that were actually available and used in the experiments. However, less expensive machines could have been used, had they been available, without significant impact on the timings obtained. For example, in the "127 Sparc+RS6000" configuration, the 9 RS6000/550 systems could have been substituted with 10-12 RS6000/320 systems, reducing the total configuration cost by nearly $400,000! Secondly, the prices used in the table reflect costs at the time of purchase, i.e. 1-2 years earlier. In the meantime, some prices have fallen, while in other cases, much faster machines are obtainable at the same prices. For example, the cost of a Sparcstation SLC used in our calculations is \$4995; as of April 29, 1992, SUN-microsystems announced a price reduction on ELC, IPC, IPX and SS2 workstations, so that a 22 MIP ELC costs as little as \$2400. Consequently, using \$4995 today, we can purchase systems with at least double the CPU speed of the Sparcstation SLC.

Tables 4 and 5 show the MIPS and MFLOPS ratings for each of the experiments, or equivalently, each of the machine configurations used. As previously mentioned, two important factors should be considered when interpreting the significance of these results. First, the algorithm is inherently non-parallel, and highly non-local in terms of memory references. Therefore, the overall rate of instructions executed per second is (not surprisingly) low, since high rates usually require very high cache-hit percentages, as well as the potential for instruction pipelining. The second point to be noted is that for this application, integer operations outnumber floating point operations by a factor of 10:1. Further, the floating point operations are unevenly and sparsely distributed among the integer computations. The combination of these factors leads to low MFLOP rates, especially notable in the timings on architectures such as the IBM RS 6000 and Intel i860. However, we reiterate that *given the nature of the problem and the above constraints*, the *comparative* timings of this application under EcliPSe do indeed exhibit very high performance.

The MIPS and MFLOPS entries in Tables 4 and 5 were obtained by first executing the equivalent (i.e., identical) runs on the CRAY, and obtaining instruction counts for the execution path of each sampler. These counts were divided by the elapsed times measured in the EcliPSe environments to obtain MIPS and MFLOPS ratings for these configurations. Table 4 shows that the MIPS ratings achievable under EcliPSe are far superior to corresponding CRAY Y-MP measurements. The 127-processor configuration, solely using networks of workstations, achieved 1255 MIPS; nearly thirty times the CRAY rating. More significantly, a 16-processor network of Sparcstations could achieve 86 MIPS, twice the figure for a CRAY. The price-performance ratios are even more remarkable. Table 4 shows that the best price-performance was achieved by a collection of 16 RS6000/320 workstations that obtained 3.14 GIPS per \$Million. The last column is a "cost-effectiveness ratio", obtained by dividing MIPS/\$Million for an EcliPSe configuration by the corresponding figure for the CRAY. We assume that the nominal cost of a CRAY Y-MP with 8 processors is \$20 Million, with an equivalent per processor cost of \$2.5 Million. For example, the second row of Table 4 implies that a network of 16 RS6000/320 workstations is 175 times more cost effective for the integer portions of this application than a CRAY.

Table 5 shows similar factors of improvements for EcliPSe over the CRAY, somewhat attenuated in a few instances. Again, collections of workstations exhibit the best performance per processor, as well as price-performance ratios far better than either the CRAY or a hardware multiprocessor, the Intel i860 hypercube. For this application, 16 RS6000/320 workstations achieved 403 MFLOPS/\$Million, as compared to 2.29 MFLOPS/\$Million for the CRAY. The tables also indicate that effective scaling to hundreds of processors of similar capabilities is feasible. As far as the authors are aware, this is the first experiment that has utilized, interactively and simultaneously, (a) 128 geographically dispersed workstations; (b) 2 geographically dispersed 64-node hypercubes; and (c) collections of workstations and hardware multiprocessors, as *a unified concurrent machine*.

# 7 Conclusions and Discussion

## Acknowledgements

We gratefully acknowledge the cooperation of various friendly machines at the California Institute of Technology, Florida State University, Oak Ridge National Laboratory, Purdue University, and Emory University, and the CRAY Y-MP at the University of Illinois. We thank Ling-Yu Chuang for her help with the CRAY measurements.

## References

[1] Bailey, D., "How useful are Today's Parallel Computers?," *Computers in Physics*, Vol. 6, No. 2, Mar/April 1992.

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

[3] de Gennes, P. G., *Scaling Concepts in Polymer Physics* (Cornell University, Ithaca, N. Y., 1979) and references therein.

[4] Flory, P. J., *Statistical Mechanics of Chain Molecules* (Interscience, New York, 1969).

[5] Harris, A., B., J. Phys. C **7**, 1671 (1974), and A. B. Harris, Z. Phys. B **49**, 347 (1983).

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

[7] Hoshen, J., and Kopelman, R., Phys. Rev. B **14**, 3438 (1976).

[8] des Cloizeaux, J., and Jannink, G., *Polymers in Solution: Their Modelling and Structure* (Oxford, New York, 1990) and references therein.

[9] Kremer, K., Z. Phys. B **45**, 149 (1981).

[10] Lam, P. M., J. Phys. A **23**, L831 (1990); but see also H. Nakanishi and S. B. Lee, J. Phys. A **24**, 1355 (1991).

[11] Lee, S., B., and Nakanishi, H., Phys. Rev. Lett. **61**, 2022 (1988), and S. B. Lee, H. Nakanishi and Y. Kim, Phys. Rev. B **39**, 9561 (1989).

[12] Le Doussal, P., and and Machta, J., J. Stat. Phys. **64**, 541 (1991).

[13] Meir , Y., and Harris, A., B., Phys. Rev. Lett. **63**, 2819 (1989).

[14] B. Mandelbrot, *Fractal Geometry of Nature* (Freeman, New York, 1982).

[15] Nakanishi, H., and Moon, Physica A **191**, 309(1992).

[16] Rego, V., and Sunderam, V., Experiments in Concurrent Stochastic Simulation: The EcliPSe Paradigm, *Journal of Parallel and Distributed Computing*, Vol. 14, No. 1, pp. 66–84, January, 1992.

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

[18] Stanley, H., E., *Introduction to Phase Transitions and Critical Phenomena* (Oxford, New York, 1971).

[19] Stauffer, D., *Introduction to Percolation Theory* (Taylor and Francis, London, 1985).

[20] Sunderam, V., and Rego, V., EcliPSe: A System for High Performance Concurrent Simulation, *Software, Practice & Experience,* Vol. 21, No. 11, pp. 1189–1219, November 1991.

[21] A. Kolawa, The Express Programming Environment, *Workshop on Heterogeneous Network-Based Concurrent Computing*, Tallahassee, October 1991.

[22] D. Gelernter, Domesticating Parallelism, *IEEE Computer*, 19(8):12-16, August 1986.

[23] V. S. Sunderam, PVM : A Framework for Parallel Distributed Computing, *Journal of Concurrency: Practice and Experience*, Vol. 2, No. 4, pp. 315-339, December 1990.

[24] High Performance FORTRAN Forum, *High Performance FORTRAN Language Specification*, Technical Report, Rice University, Houston TX, 1992.

[25] M. Rinard, D. Scales, and M. Lam, Heterogeneous Parallel Programming in Jade, *Proc. IEEE Supercomputing '92*, Minneapolis, pp. 245-256, November 1992.

[26] A. Lenstra and M. Manasse, The Number Field Sieve, *Proc. Symposium on the Theory of Computing*, Baltimore, May 1990.