The Cycle Server: A Web Platform for Running Parallel Monte Carlo Applications on a Heterogeneous Condor Pool of Workstations

Mike Zhou
Program in Scientific Computing
University of Southern Mississippi
Hattiesburg, MS 39406–0057 USA
mhzhou@us.oracle.com

Michael Mascagni
Department of Computer Science
Florida State University
203 Love Building
Tallahassee, FL 32306–4530 USA
mascagni@cs.fsu.edu

Abstract

We have developed a scientific computing tool which we call the cycle server. This tool enables Monte Carlo computations on the Condor cycle scavenging system. The tool presents users with a graphical web interface which hides the complexity of distributed computing from users. Users need only upload codes and download results via the web GUI anywhere there is web access. Parallel random number support (via SPRNG random number library), running tasks on heterogeneous platforms (via the remote compiler), and using Condor to migrate jobs from busy or dead machines to idle ones, are all integrated seamlessly with the tool and are transparent to the user. At the same time, the cycle server acts as a gate-keeper for the workstation pool. No user accounts are needed on pool computers, and jobs only run at otherwise idle times without accessing local file systems. We show the utility of the cycle server on a large Monte Carlo computation from biochemical physics.

1 Introduction

Monte Carlo calculations are based on defining random variables with mean values equal to numerical quantities of interest. As such, Monte Carlo computations have errors which are probabilistic in nature, and the error size is typically expressed through the well-known statistical quantity called the standard error. One typically assumes that the problem variance, σ^2, is constant and independent of the number of observations. Then there is a square law relationship [4] between the standard error of a computation, $\sigma(n)$, and the number of observations, n:

$$\sigma(n) = \sigma/\sqrt{n}. \quad (1)$$

One can see that to reduce the error $\sigma(n)$ ten-fold calls for a hundred-fold increase in the number of observations. Due to this fact, Monte Carlo calculations tend to converge slowly and are often compute bound. In fact, it has been said that Monte Carlo computations have consistently consumed over half of all U.S. Department of Energy supercomputer cycles for as long as the Department of Energy has had supercomputers: Monte Carlo is insatiable.

Over the years, raw aggregate computing power of uniprocessors has increased steadily *a la* Moore's law. During this same period, a sizable portion of the high end of computing power has shifted from mainframes and supercomputers to workstations and PCs. The software tool we describe here utilizes the Condor software system for the underlying resource management of a pool of workstations to provide these distributed cycles for Monte Carlo.[1]

Condor harnesses idle cycles on workstations by managing daemons on these machines. Condor jobs are "submitted" to a machine that then farms out jobs to a pool of available machines within a large, heterogeneous Condor "flock." The three main activities that Condor supports are (1) matchmaking, (2) remote system calls for I/O, and (3) process migration. The resources of a Condor pool are heterogeneous, so Condor develops a match-making mechanism for resource management. Each machine and each job have certain attributes that are specified by their owners to identify job requirements and machine constraints. Condor evaluates these requirements to find a compatible match between a job and machines. Since allowing remote jobs access to a local file system is wrought with security implications, Condor uses remote system calls to redirect the job I/O to the hard disk of the submitting machine so that the hard disk of the executing machine will not be affected. The last and probably the most important Condor functionality

[1] Condor was developed by a group directed by Professor Miron Livny of University of Wisconsin at Madison [2, 6, 9].

0-7695-0771-9/00 $10.00 © 2000 IEEE

is process migration. When the owner of a machine needs her machine, Condor stops its job and continues them on another idle machine in the pool by utilizing its fairly general checkpointing mechanism.

To run Monte Carlo applications in parallel on a Condor pool, a highly reliable parallel random number generator is required to ensure convergent and reproducible results. There are basically two methods for parallelizing random number generators. The conventional method is to split up a serial generator into subsequences and distribute them to parallel processes. [1, 5]. Traditionally, serial pseudorandom number (PRN) sequences were split by either blocking or via a leapfrog technique. These method have limitations, including short substreams, correlation among substreams, and increased computational cost on large-scale Monte Carlo applications. In contrast, the Scalable Parallel Random Number Generators (SPRNG) library developed by us [7] provides parallel PRN generators via the parameterization of full-period serial PRN generators. Here users can get full-period, independent random number sequences which are ideal for large-scale parallel Monte Carlo applications. Another important feature of SPRNG, is that it includes several, qualitatively different PRN generators. There is no single PRN generator that provides bias-free random numbers to every Monte Carlo application. Thus, one must have several, distinct, PRN generators to control a computation for unforeseen interactions between a PRN generator and a Monte Carlo application.

Thus, a solution to our problem is to develop a way to gather Monte Carlo cycles with Condor using the SPRNG capabilities. To combine SPRNG with Condor, we wrote a remote compiler to permit easy access to those Monte Carlo cycles. Note that we did not create the remote compiler from scratch. It invokes a native compiler on machines acting as compiler servers, and it also links the SPRNG and Condor libraries automatically with submitted source code. This, working together with Condor, enables a user to compile and run a job on a remote machine in a Condor pool with a different architecture/operating system (ARCH/OS) type from that of his local submitting machine. The remote compiler client-server application is written in Java because of this language's "write once run anywhere" property. This is important since the application has to run in a heterogeneous environment. In addition, we used Java socket networking and invented a simple protocol for text and binary data transfer. The server employed a multi-threading technique to handle multiple client connections.

Using the SPRNG library, the remote compiler, and Condor, a user can compile and run jobs on a pool of architecturally heterogeneous workstations given that the user has an account on one of the pool workstations. She could also partition a large Monte Carlo job into smaller jobs each running on a different workstation and consuming random numbers from independent SPRNG PRN streams. However, our scientific computation tool could be made more convenient and secure by taking advantage of the world wide web. So, we developed a component called the cycle server which provides a web interface for submitting Monte Carlo jobs, via source code, to be executed on a distributed pool of workstations.

The plan of the paper is the following. In §2 we give an overview of the cycle server. In §3 we detail the cycle server's user interface and in §4 we present implementation details for this web-based tool. In §5 we present a Monte Carlo application from biochemical physics that was effectively run using the cycle server. This work is presented as an example of the benefit the tool already provides. Finally, in §6 we briefly present conclusions and mention some possible future directions for this work.

2 A Web Platform Facilitates Secure Distributed Scientific Computing

Given a pool of networked computers, one popular way to make use of their raw computing power is to remotely log onto each of the computers and run jobs individually on them. A problem with this approach is that a user needs to have an account on each machine, and must manage passwords and often complicated security procedures if the machines are located at a remote site. This is very clumsy, and there are clearly better ways to get at the idle cycles on distributed workstations for Monte Carlo.

Naturally, accessing distributed resources over the web poses serious security issues. To address these security concerns we built a web application which we call the cycle server (Fig. 1). While the cycle server is essentially a web application, unlike regular web applications, which are for the retrieval of static or dynamic information residing or accessible from a server, our interface permits a user to submit jobs to be compiled and run on a Condor pool: users obtain CPU cycles from this server. The cycle server is made up of a group of Java Servlets together with dynamic HTML pages. Ideally, this system would run on a secure web server connected with the Condor pool via a direct cable connection and separated from the pool via a fire wall. However, in our initial implementation, this is not the case.

A user can submit a job to the cycle server from anywhere there is access to the Internet via a web browser. After initial access, the machine providing the browser can be disconnected from the internet. When the submitted job is finished, the user is informed by e-mail and the results can be downloaded, again using a web browser. The web interface both insulates and connects the user and the cycle source: the Condor pool. Viewed as an insulator, the cycle server prevents the user from directly accessing the pool machines by using, for example `rlogin` or `telnet`. This

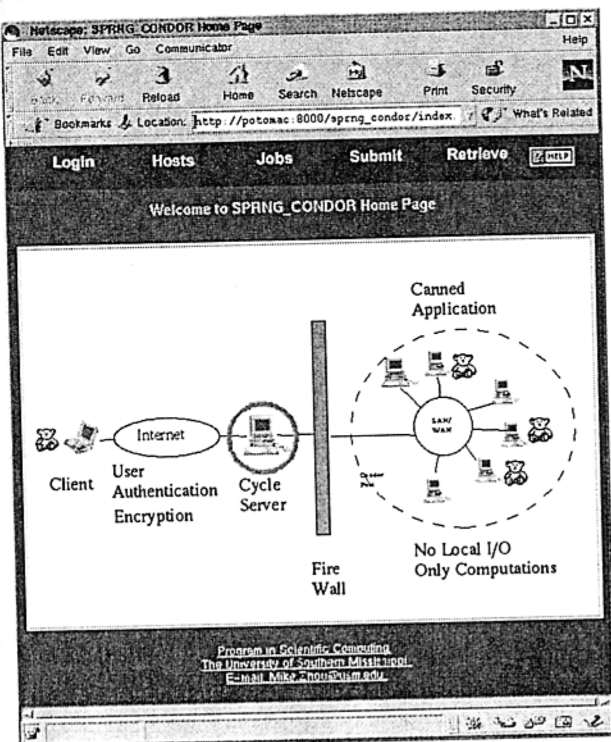

Figure 1. SPRNG_Condor cycle server home page

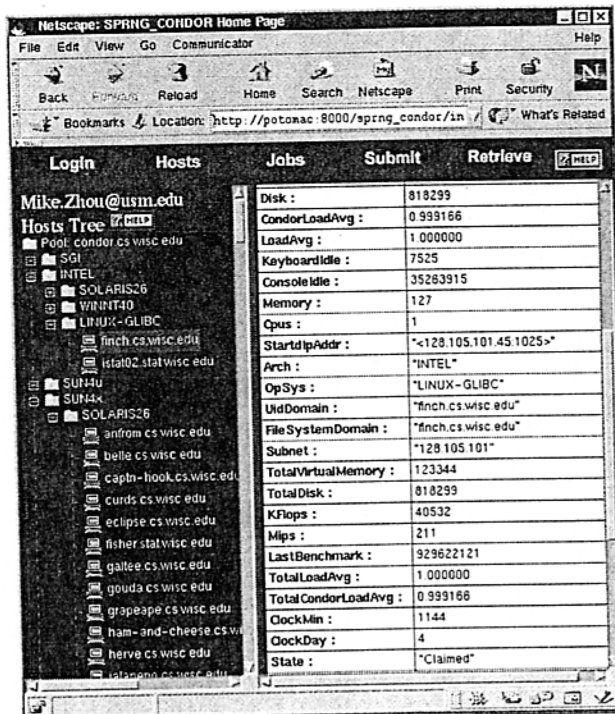

Figure 2. Host tree and host information

protects the pool computers. On the other hand, the interface also facilitates user interaction with the Condor pool. Running a job is as easy as uploading codes and downloading results from the web.

3 The User Interface

The user interface is made up of a series of web pages each of which has a menu bar on the top of the page and a display area below this bar (sample snapshots are in Fig. 1 through Fig. 4). There are five buttons on the menu bar: "Login", "Hosts", "Jobs", "Submit" and "Retrieve". Figure 1 shows a snapshot of the first page displayed when a user visits the SPRNG_Condor cycle server web site at

"`HTTP://potomac.st.usm.edu:8000 /sprng_condor/index.html`".

To log onto the cycle server, a user must have an account consisting of an email address (as a login name) and a password. After login, a user can click on the "Hosts" menu button to view the pool machine information. The user will get two frames below the menu bar (Fig. 2). The left frame contains an embedded Java Applet which organizes the machines into a tree format. By clicking our way down the host tree we can get real-time machine and machine group information which is useful in deciding which group or groups to which a job should be submitted. In Fig. 2, detailed information about the pool machine "finch.cs.wisc.edu" is displayed in a table in the right frame which we obtain after we click on the machine name.

Similar to the "Hosts" page, the "Jobs" page displays information on the pool's Condor jobs.

The "Submit" button in the menu bar helps users upload source files, compile, and run them on the Condor pool machines (Fig. 3). The display area under the menu bar is split into two frames. The left frame has a small form for users to upload source files and input files to the server, while the right frame lets a user provide compilation and execution information. Typically a user will need to upload one `Fortran` or `C` source file and some input data files. The user could fill in the form with the full paths of the files or browse and find each file and upload them one at a time. The button labeled "Browse..." implements the later.

When a job is finished the submitter will be notified via e-mail. The submitter can then download the output of the job using the "Retrieve" menu button. Using this button, a user can also check the status of running jobs, and stop or delete jobs at any time.

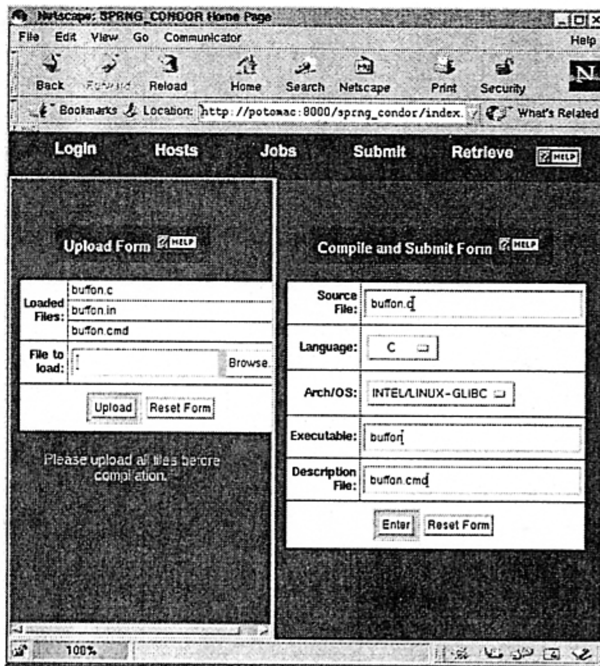

Figure 3. Forms for submitting a job.

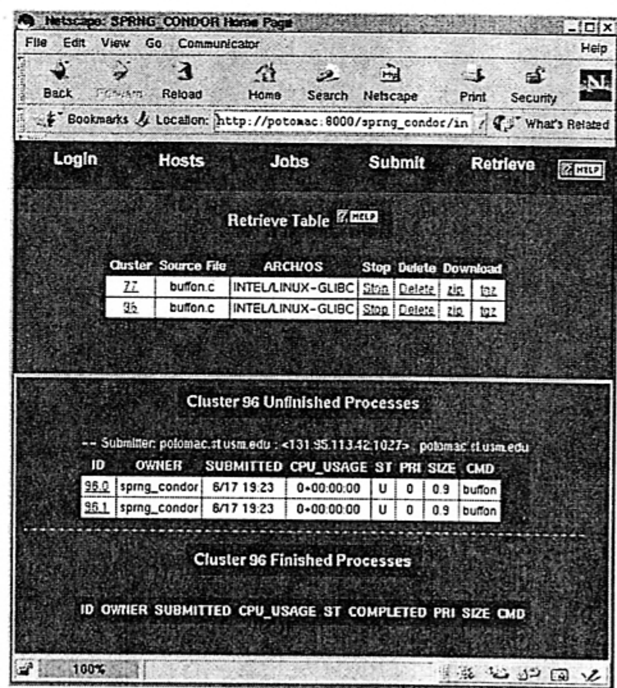

Figure 4. The retrieve page.

4 The Implementation

The cycle server application could be set up on any web server which supports Java Servlets. We currently use the JRun Servlet engine and the JRun web server. The cycle server files are contained in three directories: "sc_htdocs", "sc_Servlets" and "sc_data". The directory "sc_htdocs" contains the static HTML pages, graphic files, and Java Applet files. These are usually referred to as client side programs. The "sc_Servlets" directory contains the Java Servlets for the cycle server, and they are usually referred to as server side programs. The third directory, "sc_data", is the working place for the Servlets, it contains the cycle server password file and cycle server user home directories.

Before setting up the cycle server, one needs to first install and configure a Condor system, remote compiler clients and servers, and the SPRNG random number library. To set up the cycle server with JRun, one needs to put the three cycle server directories into their proper JRun directories and start up the JRun service manager. This, in turn launches the JRun web server and Servlet engine. We wrote a shell script to let the operating system startup and shutdown the JRun service manager automatically at machine boot and shut down time. Our implementation was made on dual processor Pentium-Pro machine running Redhat Linux.

4.1 User Authentication and Session

A networking protocol is said to be stateful if it maintains a state (a set of variables) between multiple requests by the same client. An example of such a protocol is FTP. The opposite is a stateless protocol such as HTTP where requests from the same client are considered to be independent of each other. Our application needs to be stateful since we don't want a user to have to repeat the authentication procedure for each new request. A "state" will be maintained in this application by using the Java Servlet's session tracking technique. Requests, which are processed by Servlets, from one client are connected to each other by a "session" object shared by all Servlets. State variables, such as a client's login and password are stored in the object so that a client can login once for the entire session. Authentication is done in the "`Login`" Servlet which stores the login information in a session object while other Servlets check the session object for authentication. A Servlet accesses or creates a session object using the following statement:

```
HttpSession session = req.getSession(true);
```

If the returned argument is "true," it will create a session if there is no session for this client yet. The "`session`" object is unique for each client.

When a user clicks on the "Login" menu button in the user interface, the "Login" Servlet is invoked. This Servlet interacts with user by generating dynamic web pages. When the Servlet is first invoked, it returns a login form for the user to fill out. The Servlet then retrieves the login and password values from the submitted form using the "`getParameter`" method of the "`HttpServletRequest`" object. The Servlet then validates the user-submitted e-mail address and password by matching them with the entries of the password file.

4.2 Viewing Real-Time Job and Machine Information

Four Servlets work together to allow a user to view the Condor pool machine information. When a user clicks on the "Hosts" menu button, the "Hosts" Servlet is invoked and generates an HTML frame set page which contains a left frame and a right frame. The graphic tree in the left frame is drawn by a Java Applet returned by the "Hosttree" Servlet.

The tree Java Applet comes from the server and runs on the client machine. It can produce graphic objects such as lines, include external images, and perform other graphical functions. It can also respond to user interactions. An Applet is an effective way to maintain such a graphic user interface. The tree Applet is created by the Servlet "HostsTree" which queries the pool in real time. Condor provides commands like "condor_status" to produce output pool machine information in text format. The Servlet "HostsTree" invokes these commands, gets the output, parses the output and uses the results to inform the Applet to construct a tree.

When a node of the host tree is clicked, "HostDetails" invokes the Condor command "condor_status," with appropriate arguments, to obtain the desired real-time information. For example, by clicking on an OS node, one will receive information including how many machines running the requested OS are idle. Clicking on a machine node will display information such as how much memory the clicked machine has.

The job information is retrieved and displayed in a similar manner to the way the machine information is displayed except that a different Condor command, "condor_q," is used by the Servlets to get submitted job information.

4.3 Submitting Jobs

Uploading a file via the web is a tricky task, and most web applications only allow a user to type in or paste text into a highly constrained form. Clearly, one reason for this is the security issue associated with uploading data and therefore permitting a remote machine access to a local file system. For the user's convenience, we allow users to upload files directly from their local file systems. To enable this feature, special care must be taken on both the client and the server side. On the client side we use an encryption protocol. Specifically, we set the encryption type to be "`MULTIPART/FORM-DATA`":

```
<form    method=POST
         ENCTYPE=
         "MULTIPART/FORM-DATA"...
```

and then we insert a "file" form field as follows:

```
<input   type=FILE name=sourceFile ...
```

To receive files, the server side Servlet "SubmitUpload" creates an `inputstream` object from the `request` object. It then reads a stream of bytes from the `inputstream` from which it obtains the file size and file name information. By stripping heading and trailing bytes, the Servlet reads and saves the uploaded file on the server's hard disk.

For each user, the Servlet creates an individual home directory under which each job occupies a separate subdirectory. User codes and other files are uploaded to the job directory. In this Servlet, we create a hash table named "jobs" for managing a user's jobs. Each entry of the "jobs" hash table has a job number as key and a "job" hash table object as value. Before and after a transaction, the "jobs" hash table is read from and written to through a job database file. This way the job information is made persistent and this task is easily carried out using the `objectinputstream` and `objectoutputstream` classes.

After a user has uploaded all the files for a job, a remote compiler client connects with an appropriate remote compiler server. That server resides on a machine with the user specified ARCH/OS type. On that server, the job is automatically compiled and linked with both the SPRNG library and the Condor library. The linked executable is then returned to the user job directory, and the executable is then submitted to ARCH/OS-appropriate Condor pool for execution.

4.4 Retrieving Jobs

When a user issues the download commands by clicking "zip" or "tgz" in the "Retrieve" page, an entire job directory is packed into one file and sent back to the user via the web. To save Internet bandwidth and download time, the executable is excluded from the job package.

5 Brownian Langevin Simulation of Large Molecules Using the Cycle Server

In this section we investigate the impact of the pseudorandom number generators on a Monte Carlo simulation of

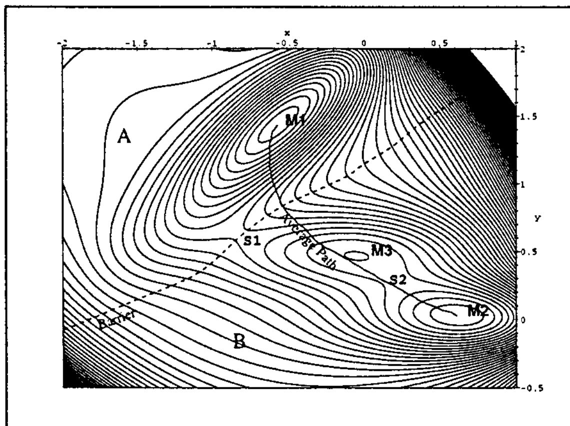

Figure 5. A Müller-Brown potential surface. The barrier (dashed curve) divides the space into two parts: part A has minimum M1 and part B has M2 and M3 as minima. The primary saddle point S1 is on the barrier curve and S2 is between M2 and M3.

the movements of large molecular systems. These calculations were done using the tool described in the previous sections and is presented here in brief detail as a testament to the utility of the tool to facilitate distributed Monte Carlo computations.

The performed computations involve both dynamics (molecular dynamics) and statics (Monte Carlo), and are of interest to medical researchers who need to predict reaction rates from potential energy profiles (c.f. [10]).

We consider the problem of finding the average path of a molecule on a Müller-Brown potential surface (Fig. 5) [8] as the molecule moves from one state stable to another. The molecule is in solution and thus subject to high-friction, and its movements are either "diffusive," "over-damped," or "Brownian." The equation of motion for the molecule is the Brownian Langevin equation:

$$\frac{\mathrm{dx}}{\mathrm{d}t} = \frac{\mathrm{f}}{m\gamma} + \mathbf{R}(t) \tag{2}$$

where $\mathbf{x}$ is the position of the molecule. The right hand side has two terms. The first term describes the deterministic motion of the molecule with mass m under the external force $\mathbf{f}$. The second term describes the stochastic part of the motion with $\mathbf{R}$ representing a random force acting on the molecule to account for the stochastic interaction with surrounding small molecules. $\mathbf{R}$ is proportional to temperature T so that at low temperature the first term in the right hand side of Eq. (2) will dominate and the motion of the particle is deterministic. At high temperature the second term in the right hand side of Eq. (2) will dominate and the motion will be totally stochastic.

A simple approach to simulating the above diffusive equation of motion, Eq. (2), is to use a simple forward-Euler discretization:

$$\mathbf{x}_{j+1} = \mathbf{x}_j + (\mathbf{f}_j/m\gamma)\Delta t + \Delta\mathbf{x}_R, \tag{3}$$

where $\mathbf{x}_j = \mathbf{x}(j\Delta t)$ and $\mathbf{f}_j = \mathbf{f}(\mathbf{x}_j)$, and $\Delta\mathbf{x}_R$ is chosen from a Gaussian distribution with variance given by:

$$\sigma^2 = 2\Delta t k_B T/m\gamma. \tag{4}$$

This simplistic approach to simulating the molecular movements via Eq. (3) is quite inefficient because only a tiny percentage of the trajectories generated will reach the final state of interest, B, in a computationally reasonable amount of time. In such situations, one can use the method of importance sampling to force all trajectories to the desired final state, [4]. Formally, one begins by constructing an integral representation of the straightforward simulation and then find a good "sampling bias", D, which is used for importance sampling. The biases we will use in this work [3] consist of products of Gaussian probability densities, each of which is centered about a space-, time-, and force-dependent target point. However, for our purposes in this paper, more detail on the exact nature of the importance functions used is not necessary.

5.1 Running the Application in Serial

Now we detail the the steps taken to investigate the behavior of the previously described Monte Carlo application as we systematically vary the SPRNG generator used. The tool we have described greatly facilitated our ability to set up and execute these calculations. In these computations we use the **standard** interface of SPRNG in the application code, which is written in `Fortran77`.[2] To use SPRNG we must first include the SPRNG header file and declare the necessary SPRNG variables:

```
#include "sprng_f.h"

       integer GEN, stream-
        num, nstreams, seed

       SPRNG_POINTER stream
```

Here "`GEN`" is the generator type, "`nstreams`" is the total number of PRN streams needed, "`streamnum`" is the stream this run will use, and "`seed`" is the global random number seed. In our computational experiments we are

[2]For a more careful description of how to use SPRNG in a Monte Carlo application, the reader is referred to the SPRNG home page: `HTTP://sprng.cs.fsu.edu`. This web page has detailed documentation, examples, and downloadable source code for SPRNG 2.0.

planning to run our Monte Carlo application with many different SPRNG PRN generators. We can do this simply by varying GEN. We then read in the parameters and initialize a random number stream as follows:

```
stream   = init_sprng(GEN, stream-
         num,nstreams,seed,SPRNG_DEFAULT)
```

The pointer to the random number stream, "stream", is used each time the program needs a new random number from this particular stream. For example, calling the SPRNG function "sprng(stream)" returns a double precision random number uniformly distributed in [0,1) from PRN stream, "stream."

```
sprng(stream)
```

To submit a job to the Condor system, one needs a description file. The following is a sample description file for this particular computation:

```
executable = lang_sprng
log        = lang_sprng.log
input      = g0_s0.in
output     = g0_s0.out
queue      1
```

This description file instructs Condor to run the executable "lang_sprng" and write log messages to "lang_sprng.log". The Condor job reads from file "g0_s0.in" for its inputs and writes to "g0_s0.out". In addtion, the cycle server will add some lines to the description file as follows:

```
notifyuser = Mike.Zhou@usm.edu
requirements  = Arch == "INTEL" && OpSys
        == "SOLARIS26"
```

The extra lines are needed so that the job will be compiled and executed on the right machine group and the cycle server user will be notified upon job completion.

With the Fortran source code, the Condor description file, and the input file, we log on and submit the job to the cycle server through the web interface. At this point we can logout and await an e-mail notification of job completion. Then we can retrieve the computational results through the same web interface.

Behind the scenes, the WB-GUI agent on "HTTP://potomac.st.usm.edu" retrieves the source file and contacts the remote compiler server on "vulture.cs.wisc.edu" informing it to compile the source code. The remote compiler will also link the code with both the SPRNG and Condor libraries suitable for the ARCH/OS pair of the remote compiler. Then the WB-GUI agent submits the executable on behalf of the use and

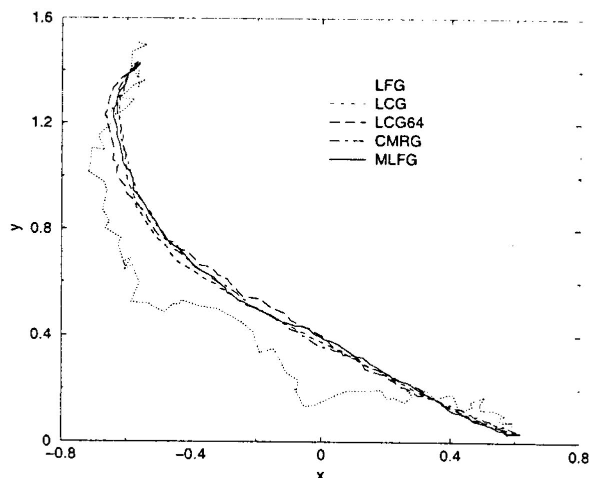

Figure 6. Average reaction paths generated by different random number generators.

the code executes on all available"INTEL/SOLARIS26" machines.

In this computation, we test five SPRNG generators with the Langevin dynamics application. The five generators used were an additive lagged-Fibonacci generator (LFG), a prime modulus linear congruential generator (LCG), a power-of-two modulus linear congruential generator (LCG64), a combined multiple recursive/linear congruential generator (CMRG), and a multiplicative lagged-Fibonacci generator (MLFG). We let the program read in a generator type number, GEN, from the standard input. This permits the SPRNG initialization routine to use the "type number" to create a PRN stream of the corresponding type.

Fig. 6 illustrates the resulting reaction paths using the five previously mentioned SPRNG generators. Here each path is an average of $3,000,000$ trajectories using the same temperature ($T = 300^oK$) and the same barrier height ($H = 7k_BT$). About 8 hours of CPU time was used for each curve on the Intel(Pentium)/Solaris machines. One can see that the curves are very close to one another except for the LFG curve. We felt that the observed deviations in the LFG case were due to fluctuations and not correlations. This hunch was later confirmed by averaging 30 times as many trajectories. Nevertheless, for this particular application we should recommend not using SPRNG generator LFG.

5.2 Running the Application in Parallel

To find out whether the deviations to the LFG curve were caused by fluctuations or intrinsic PRN correlations, we generated more trajectories and computed a new average path. We expect to reduce the statistical error five-fold

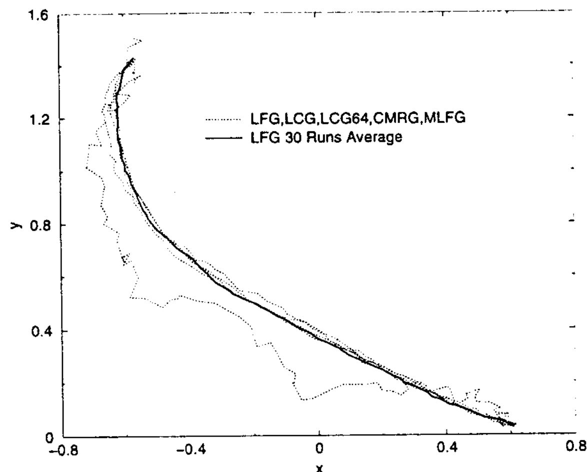

Figure 7. Comparison between 30 run average results of LFG with one run results.

if we average 30 times as many trajectories; however, this will require 10 CPU days (8 hours × 30). The cycle server makes it very easy to divide the job into smaller jobs and run them on different machines in parallel. We don't even need to change the source code. We launch 30 runs of the same program, with each run consuming a different random number stream from the same PRNG. We needed 30 input files which differs from each other in stream number, and thus wrote a simple PERL script to generate the input files. The Condor description file contains the following:

```
executable = lang_sprng

log        = lang_sprng.log

input      = g0_s$(Process).in

output     = g0_s$(Process).out

queue    30
```

Here we use one line to specify which input file each of the thirty smaller jobs uses using the macro "`$(Process)`". Each smaller job has a unique process number ranging from 0 to 29 which can be referred to by "`$(Process)`". The line "`input = g0_s$(Process).in`" states that the first run (smaller job) will use the input file "g0_s0.in", second run will use "g0_s1.in", and so on.

The 30 run average curve is plotted in Fig. 7 together with original single run curves. The 30 run average curve overlaps the others allowing us to conclude that the deviations of the single run LFG result are due to fluctuations but not to systematic correlation.

6 Conclusions and Future Work

The cycle server can be thought of as a single component of a comprehensive software tool to enable distributed Monte Carlo computations. The cycle server simplifies user interaction with a Condor pool and SPRNG. At the same time it enhances security for the distributed machines. Most importantly, this is a tool for harvesting idle cycles on distributed resources over the web. Since the target application, Monte Carlo, is naturally parallel, and SPRNG provides reproducible random number streams, we envision such a system replacing the supercomputer requirement for many large-scale, high fidelity Monte Carlo computations. In the future we plan to extend the cycle server concept to harvest cycles over the web on machines that allow a special client to run when they are otherwise idle.

References

[1] S. Anderson. Random number generators on vector supercomputers and other advanced architectures. *SIAM Rev.*, 32:221, 1990.

[2] J. Basney and M. Livny. Deploying a high throughput computing cluster. In *High Performance Cluster Computing*, volume 1. Prentice Hall, 1999.

[3] M. Z. Daniel and T. B. Woolf. private communications, 1999.

[4] J. M. Hammersley and D. C. Handscomb. *Monte Carlo Methods*. John Wiley & Sons, 1964.

[5] P. L'Ecuyer and S. C. Cot'e. Implementing a random number package with splitting facilities. *ACM Trans. Math. Soft.*, 17:98, 1991.

[6] M. Livny, J. Basney, R. Raman, and T. Tannenhaum. Mechanisms for high throughput computing. *SPEEDUP Journal*, 11(1), 1997.

[7] M. Mascagni and A. Srinivasan. SPRNG: A scalable library for pseudorandom number generation. *ACM Trans. Math. Soft.*, in the press, 2000. See also: `HTTP://sprng.cs.fsu.edu`.

[8] K. Müller and L. D. Brown. Location of saddle points and minimum energy paths by a constrained simplex optimization procedure. *Teoret. Chim. Acta*, 53:75, 1979.

[9] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[10] T. B. Woolf. Path corrected functionals of stochastic trajectories: Towards relative free energy and reaction coordinate calculations. *Chem. Phys. Lett.*, 289:433, 1998.