2003

Extensions and Optimizations to the Scalable, Parallel Random Number Generators Library

Jason Parker

FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

EXTENSIONS AND OPTIMIZATIONS TO THE SCALABLE, PARALLEL RANDOM NUMBER GENERATORS LIBRARY

By

Jason Parker

A thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Fall Semester, 2003

The members of the Committee approve the thesis of Jason Parker on November 14, 2003.

Michael Mascagni
Professor Directing Thesis

Ashok Srinivasan
Committee Member

Alec Yasinsac
Committee Member

Robert van Engelen
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

This work is dedicated to my family: my grandmother, Vonnie Wood, my parents, Fred and Lucy Parker, and my brother, Fred Parker Jr. I am eternally grateful for their love, wisdom, and support.

ACKNOWLEDGEMENTS

This document was created under the aegis and tutelage of Dr. Michael Mascagni. I thank him for all his help with this document and my education in general. I thank him also for the assistantship he provided me during my graduate study at Florida State University.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

This work will examine enhancements to the library for scalable, parallel pseudorandom number generation (SPRNG). SPRNG uses parameterization to produce many streams of random numbers with emphasis on parallel Monte Carlo methods. We extend the previous work to enable random access to these streams. This new method for generating streams improves both functionality and intuition of interface. Also considered are a few memory optimizations to the SPRNG library.

CHAPTER 1

INTRODUCTION

This work will examine computational random number generators towards the end of improving the Scalable, Parallel, Random Number Generators (SPRNG) Library. We work toward establishing a more intuitive and useful interface to extend the purposes for which SPRNG is useful. We also concentrate on reducing memory usage to provide high performance users the efficiency needed for ever larger and more parallel Monte Carlo applications (applications that consume random numbers). Monte Carlo algorithms provide more economical solutions to many problems than deterministic methods. Since many Monte Carlo applications are naturally parallel, allowing for a greater degree of parallelism is of importance to any random number generator that supports such applications. The work herein will extend and optimize SPRNG toward creating a more useful and general random number generator. We will first give a necessary background for random number generation and SPRNG.

1.1 Randomness

Randomness is a property of nature that has fascinated man since before computers were a twinkle in Turing's eye. We have used it as an essential part of our sports and games, settled disputes with it at times, and even fought it in our weak predictions of the near future. The apostles even used a random source, casting lots, as an interpretation of the hand of God in choosing the apostle to replace Judas. While computers don't gamble or worry about the weather, randomness appears to be a valuable resource in computing in the general, but the sorts of randomness a computer can use tend to be more specific. Computers are in fact deterministic, at least as much as we can make them. We try to make the probability of errors in our computations as small as possible, and make great efforts to correct random faults that do arise. While physical sources of randomness provide an intuitive solution, this field has met with much opposition, with even the most sophisticated methods yielding complex patterns that betray the randomness they hope to provide. So until a source of randomness is identified, perfect not only in its statistical properties, but also in the efficiency with which it produces random numbers, programmers settle for only a partial definition of randomness when compared with the intuitive nature of non-determinism. However, we will see that not much is lost if deterministic methods for generating random numbers are thoroughly evaluated and tested against the many properties we have found them to possess.

1.2 Physical Sources of Randomness

In understanding the properties that a sequence of numbers must possess in order to be used in place of a random sequence, it may help to first examine the intuitive nature of randomness that arises all around and why such a natural source has not come to widespread use. One of the most popular examples for addressing this issue has always been games of chance. Consider the roulette wheel. For those not familiar with the game of roulette, it consists or a flat wheel with 37 or 38 slots on top. While the wheel is spinning, a metal ball is thrown along the outside of the wheel and eventually settles into one of the slots. This might be one of your first candidates to add to a computer as a physical source of randomness. The ball and wheel could be set in motion by simple mechanical design. Sensors in the slots could then give the computer a random number in [0,N) where N is 37 or 38 depending on the wheel.

However, scientists studying chaos theory found a way to beat the roulette game, which would be impossible if every spin gives a truly random number. They were able to measure the speed of the ball and wheel while a certain dealer was working the apparatus and predict that some numbers had higher probability than others [1]. Also consider the time it takes for the ball to settle into a slot. It takes so long, that casinos allow betting during the early part of the spin. Both these serve to make a roulette wheel an impractical random number generator. If an operator disturbs the probability distribution of the wheel by tending to spin the wheel and throw the ball at a certain speed, certainly a mechanized version would have the same problem. While many sources of randomness have errors in their probability distribution functions usually small enough for us to deal with, it is the inefficiency of these physical sources that have kept them out of wide spread use in computing.

Another attempt for physical generation has been based on radioactive decay. The problem that arises in this technique is the imprecision of measurements involved. The sort of detector used becomes less sensitive to decay for a brief period following detection, thus allowing small errors to arise [12]. Similar problems with measurement arise in some other suggested physical generators. In fact, most random numbers based on the measurement of physical randomness are of poorer quality than the simplest pseudorandom number generators.

Yet another reason that physical processes do not provide a good source for randomness in scientific applications is that the numbers are not reproducible. A key element of an objective scientific experiment is that the results can be verified by repeating the experiment; this quality is called reproducibility. The numbers produced by a physical source of randomness cannot be guaranteed to be identical to any previous recorded observation. Reproducibility is especially important where computer programming is involved. If a bug is identified that depends on the random numbers a program uses, it is necessary to observe those conditions again to confirm that the bug is removed. So we turn to generating our sequences of numbers arithmetically, and hope to preserve properties of our numbers that will be useful in the computations to be performed.

1.3 Pseudorandom Number Generators

A pseudorandom number generator is a deterministic recurrence that passes tests of randomness and performs well in applications requiring randomness. Take as an example the Linear Congruential Generator (LCG). The LCG uses a recursive method to generate a random sequence and is defined by [9]:

$$X_{n+1} = (aX_n + b) \bmod N$$

where X_n represents the nth number of the sequence, a is the multiplier, b is the additive constant, and N the modulus. X_0 is called the seed, being the initial value for the recurrence. The values that do not change in the recursion are called parameters. This is one of the most popular generators today and three of the six generators in SPRNG are versions of the LCG. In order for this recursion to produce values that are random, the parameters must be chosen properly. There are many properties of a random number generator that combine to make it resemble a random sequence, not the least of which are period and distribution.

Periodicity is an important property of any recursive random number generator. Since a random number generator is stored in a finite amount of memory, it must eventually come to a state that it has been in before, after which it will begin to repeat the same numbers. The amount of numbers produced in this repetition is the length of the period. That is, the period of a sequence is the smallest ***p*** such that:

$$\exists n_0 \text{ such that } \forall n > n_0, X_{n+p} = X_n .$$

Random number users must be careful not to use numbers after the period has expired. Thus, it is important to know exactly how many numbers will be produced by a generator before it begins to repeat itself. Running over the period length can be detrimental to many random number applications. Take for instance the LCG. The properties necessary for maximal period depend on the form of the modulus chosen. SPRNG provides both a power-of-two modulus version and a prime modulus version. With symbols preserved from the above equation, the power of two modulus version has a period of N when:

1. $N = 2^k$
2. $k > 2$
3. $a \equiv 3 \bmod 4$

The prime modulus version has a period of N-1 whenever the multiplier is a primitive root of the integers modulus N. Since N is prime, this means that the order of the multiplier must be N-1. The order of an element α modulus M is the least β such that:

$$\alpha^{\beta} \equiv 1 \bmod M$$ [2].

Since the period is N-1, one element of integers modulus N must be omitted from the sequence. This element is given by:

$$X = b(a-1)^{-1} \bmod N$$

Another important property of random number generators is distribution. The most commonly used is the uniform distribution on [0,1). Thus, in practice, the numbers produced by the LCG are divided by the modulus to map them onto this interval. A very good measure of the distribution of the LCG is the spectral test to be described later.

Even with maximal period and a good distribution, our sequence goes only so far. Take for instance choosing a =1 and b =1. Any N can be chosen to meet the requirement for maximal period, because the sequence produced is the integers modulo N in order; this is not very random. Surely sequential numbers are not very random. To further ensure quality of random numbers, they must pass many statistical tests. It is widely known that no one test that suffices for a general random number sequence. In fact, for a given application different tests become more or less important because of the properties desired of the numbers for that particular application. Strong mathematical evidence for good performance on these tests based on the properties of a generator serves as a credible source to augment the empirical evidence provided by computing the tests on portions of the random number sequences.

The first discussed is the Chi-square test [8]. Consider an apparatus designed to yield 1 of *n* outcomes with p_j being the probability of the j^{th} outcome. The apparatus is also designed to yield an independent result on each of its activations. The Chi-square test described by Knuth[1] gives a way to test the validity of such an apparatus. This test is among the most popularly used today. The idea is to use our apparatus a large number of times and calculate the difference of the observed distribution and the expected distribution. Should the apparatus be used *M* times, the j^{th} outcome is expected to occur $M * p_j$ times. Let R_j be the observed number of times the j^{th} outcome arises. We the sum the squares of the difference of R_j and $M * p_j$. We must also normalize to account for the differing probability of outcomes and after some manipulation arrive at the formula:

$$X^2 = \sum_{j=1}^{n} \frac{(R_j - M * p_j)^2}{M * p_j}$$

X^2, called the Chi-Square statistic, is then compared to a distribution based on the value *n-1*, called the degrees of freedom, giving us an idea of how far the value is from average. A value that is too high or low suggests a sequence may not be a good source of randomness.

So far we have described here a hypothesis test for a random source with a finite number of outcomes. Extending the idea to a continuous version, we can test a random number generator on [0,1) (the real interval greater than or equal to 0 and less than 1) by dividing that interval into bins and counting the numbers in each bin. We can also choose a more complicated outcome based on several random numbers and apply the Chi-square test. For example, the poker test takes *n*-tuples of random numbers and determines their relative ordering so that one of the *n!* permutations of the ordering can be identified.

Then *n!* bins are used to tabulate the empirical distribution of the permutations after which the Chi-square test can be used to check whether this distribution is uniform.

While the Chi-Square test is designed for testing the quality of a discrete probability distribution, the Kolmogorov-Smirnov (K-S) test [8] is designed to check continuous probability distributions. The idea is that for any value, x, in [0,1) we can calculate the expected number of sequence elements that are less than x. We can then calculate the maximum difference between this expected value and the observed value to determine how much they differ statistically. This maximum is easy to compute because we only need to measure the difference at each point produced by our sequence. The sequence can be sorted into ascending order and each considered in turn. Much like the Chi-square test, the expected distribution for these measurements can be calculated based on the number of elements in the sequence to be tested.

One of the most powerful tests used today is the spectral test [8]. The ideas underlying it are more complicated that those of the aforementioned tests, and obtaining the results requires more computation. The idea behind the spectral test is to examine overlapping tuples within a sequence. Say we decided to use a 3-dimensional spectral test; we would consider elements 1,2,3 of the sequence then 2,3,4 and so on. The space the random numbers fall along is then divided up into hypercubes and a count is made for each section. The higher the dimensionality of the test, the more sections the space is divided into. Good random number generators should perform well in dimensions numbering at least 6 to 10. For example, these tuples reveal something very interesting about the Linear Congruential Generator (LCG). The tuples produced by the LCG will lie in parallel hyperplanes [10]. For instance, 2-tuples would lie on parallel lines in two dimensions and 3-tuples would lie along parallel planes in three dimensions. This test can be applied to the output of an LCG to determine the quality of its multiplier. Moreover, applying the spectral test to an LCG allows number theoretic shortcuts for calculation to reduce computational intensity.

While the structure of the tuples of an LCG looks suspect, many multipliers can be used to pass this test in several dimensions. This is a good example of how many statistical tests are necessary to ensure proper quality of generators, with differing applications better served with differing generators. The programmer of an application should be well informed about the statistical properties of particular random number generators when deciding which one would best suit the application. The notion of uniform distribution given by the spectral test is a powerful one however. Many Monte Carlo applications, such as multidimensional integration, converge much faster with uniformly distributed numbers, even when other properties are neglected. This has given rise to quasirandom numbers, which prioritize uniform distribution above all other properties for those applications in which they yield faster convergence [13].

The architect of a random number library must provide various methods to meet the needs of a large audience of random number consumers. While SPRNG has generators that meet the requirements for many different statistical tests and measurements, many opportunities for broadening the application of the library are available. An important consideration is the lack of knowledge most users have about the specifics of random number generation. Good information hiding practices require that our library should be available for use with as little knowledge of the underlying architecture as possible. These ideas and many more have been taken into to consideration to design SPRNG, and we hope to improve on them here.

CHAPTER 2

THE SCALABLE, PARALLEL, PSEUDORANDOM NUMBER GENERATORS (SPRNG) LIBRARY

The SPRNG library is written in the C programming language, with interfaces added to support the FORTRAN programming language. It has been designed to provide support for a wide range of random number needs, from a simple interface for those users who just need a little randomness, to generators yielding billions of independent random number streams. The remaining portion of the document will be applicable to the standard SPRNG interface and is geared toward high performance, parallel applications.

2.1 Parallel Random Numbers

The tests above apply to testing the output of a single, serial random number generator. It has been repeatedly observed that Monte Carlo applications (those applications that use random numbers) are often naturally parallel. Many of these applications make many runs of the same algorithm with different random numbers until the error is acceptably small. It immediately follows that being able to distribute the runs among computing nodes provides a near linear speed up in the factor representing the parallelism occurring. That is, having N available computational nodes allows for a speed up of nearly N times. In most Monte Carlo applications, little or no interprocessor communication is needed as the runs are independent by design. Many parallel applications require results computed in parallel to be available to other processes. This creates overhead similar to any networked effort. Some processes will also require information from other processes to perform, creating a *de facto* serial aspect to the application. Since many Monte Carlo applications do not need this sort of communication, they are good candidates for high performance, massively parallel calculations.

To accommodate parallel applications, more than one sequence of random numbers must be used. In this context we will refer to these sequences as streams. The reader should understand that these streams are usually from the same mathematical family, not only sharing a common method of generation, but many of the parameter choices as well. It simply will not do to have only one serial generator. Not only would this be horribly inefficient, adding an unnecessary shared resource to the mix, making it reproducible would be difficult, because the numbers would not necessarily be consumed in a particular order because of real time variations.

The reader will probably intuit that we could not use the same random number stream for each parallel task, so we turn to the ability to produce many different streams of random numbers. In fact, the statistical independence of runs in Monte Carlo applications is very much affected by the statistical quality of the streams used. We describe two methods for creating a large number of streams, namely splitting and parameterization.

The first of these, splitting, takes a single random number stream as the source for all the numbers used in all parallel processes. Splitting has many variations. For example, blocking is a method whereby one long stream of random numbers is cut into contiguous pieces. Suppose we wish to split a sequence of length *N* into *M* pieces (assume *M*|*N*), then the first stream will be:

$$X_0, X_1, \ldots, X_{\frac{N}{M}-1}.$$

The second stream will be:

$$X_{\frac{N}{M}}, X_{\frac{N}{M}+1}, \ldots, X_{\frac{2N}{M}-1}.$$

and so on. This method provides ease of generation after the stream is split, with the main difficulty in determining which initial condition to give each of the pieces so that they will be equally long. The user must take care to not let one stream run over into the next stream's numbers. Also, blocking is prone to correlations among the streams produced [3, 4]. In fact, any generator based on blocking will have strong correlations among the streams. Blocking is not used in the SPRNG library because of this generic defect.

Leapfrogging is another splitting method that can create many streams out of an original by decimating the sequence of numbers. That is, if we want to create *N* streams from one stream by leapfrogging, each sequence receives the numbers of a different congruence class modulo *N*. If X_j is the jth element of the original sequence, then the first stream will be:

$$X_0, X_N, X_{2N}, \ldots$$

The second stream will be:

$$X_1, X_{N+1}, X_{2N+1}, \ldots$$

and so on. This can be thought of as "dealing" random numbers as one deals cards for play. There has been interest to adding such a method to SPRNG to provide more streams than are currently available for some generators and this method has been tested on the SPRNG LCG generator with very good results. However, parameterization is the primary way SPRNG provides streams of random numbers for parallel applications in an efficient, reproducible way. Parameterization underlies the work done here.

2.2 Parameterization

SPRNG has six different random number generators. Each generator provides a different mathematical method for generating random numbers. Reusing a previous example, the linear congruential generator (LCG) uses the recurrence:

$$X_{n+1} = (aX_n + b) \bmod N$$

Each of the generators in SPRNG is capable of generating several streams by varying the parameters in the generator, *a* and *b* in the LCG example. These streams all must satisfy a condition of independence, that is, there is no appreciable correlation between any two streams. Much of the work on SPRNG has been done on the way users can instantiate those different streams.

Parameterization is a complex issue and one of the most important aspects of the SPRNG library. It is parameterization that provides the scalable parallelism for which SPRNG is named [11]. The difficulty in parameterization lies in the additional requirement that the individual streams must be independent, that is they may have no appreciable correlation. This is in addition to the already required properties of serial random number generators that each stream must satisfy. Furthermore, the parameterization must be easy to compute to provide efficient instantiation of streams. An appropriate form for the parameterization of LCG must be based on consideration of the form of the modulus. Parameterizations are discussed in [11] based on both prime and power-of-two moduli. Both of these are available in SPRNG through different versions of the LCG generators included.

For example, in the power-of-two version, the parameter varied is the additive constant (*b* in the above equation). Of course, not every set of constants will do; sets must contain elements that are relatively prime to all the others. In the SPRNG implementation, the i^{th} stream uses the i^{th} prime as its constant. The Fibonacci generators in SPRNG, both the modified lagged-Fibonacci Generator and multiplicative lagged-Fibonacci generator, use seeding for parameterization. In turn, the seed given by the user is combined with the parameterized seeding to yield different streams for different seeds.

For the sake of implementation, SPRNG generators impose a numbering scheme on the parameterized streams. The identification of a stream in this scheme is called the stream number. This numbering scheme may seem implicit since it takes an appropriate form for each generator, but nonetheless all of the current SPRNG generators yield to a numbering system for the different streams.

2.3 The Fibonacci Generators

Two very important generators in SPRNG are the modified lagged-Fibonacci generator (LFG) and the multiplicative lagged-Fibonacci generator (MLFG). These generators provide far more possible streams than the other SPRNG generators. This makes the Fibonacci sequences a favorite for high-performance, massively parallel applications. Many SPRNG users need more streams than are available in the other SPRNG generators. However, both of these generators use a large array to store past values, since they combine two of these numbers to calculate the next value. Instead of the number of streams available, memory becomes the prohibitive factor to the level of parallelism many SPRNG users are trying to attain. The modified lagged-Fibonacci generator was developed to overcome certain weaknesses in the additive lagged-Fibonacci generator [11]:

$$X_n = X_{n-j} + X_{n-k} \bmod 2^m \text{, where k > j.}$$

The modified version uses the exlusive-or (XOR) of two different, but related, additive lagged Fibonacci sequences [14]. The multiplicative lagged Fibonacci generator is as it its name might suggest [11]:

$$X_n = X_{n-j} * X_{n-k} \bmod 2^m \text{, where k > j.}$$

Notice that the Fibonacci generators require *k* numbers as initial values; *k* is called the lag.

The users of these generators will benefit the most from the work explained here. Because of the large number of initial values needed, these generators offer the opportunity for a large number of different seeds. The downside is that each stream will have to carry a large array of numbers already generated, consuming more memory than other generators. These generators are particularly suited for parallel applications because of the large number of streams available. However, the large state space (memory consumed by a stream's underlying data) is often prohibitive for using all of the streams available. Those with very large lags are suitable for applications requiring few streams of long period. Those with smaller lags are suitable for generating many streams of random numbers, provided the lag is not so small as to limit the period length or the quality of the streams below what a user needs.

2.4 Initialization and Spawning

SPRNG uses a two-function method for user generation of random number streams. The two functions called are *init_rng* and *spawn_rng*. The user first calls *init_rng* once per stream created and usually at least once per processor in a parallel computation. These calls are usually done in a single step at the beginning of the computation to create a random generator in each processor. Each call takes as input the number of times the *init_rng* function is to be called (*total_gen*) and the stream number (*gennum*). Error checking tests that *gennum* is in [0,*total_gen*). Subsequent to these calls, a user calls the *spawn_rng* function to produce new streams. The *spawn_rng* function takes as input a number of streams to be produced (*nstreams*) and an already existing stream in the form of the appropriate *struct*. A *struct* is a programming construct in the C language that allows many data items of different types to be stored and handled as a single unit. This particular *struct* is prepared with spawning information used to produce the correct streams in a reproducible manner.

The reproducibility requirement is the guiding principle behind most of the architecture in the spawning process. The idea is that each stream instantiated will have reference to a set of unused streams. We will call this set of unused streams the spawn pool. Whenever new streams are spawned from a particular stream (that is to say the *spawn_rng* function is called with the *struct* of that stream), that stream's spawn pool is implicitly divided up among itself and the newly created generators. To implement such a scheme, a tree is used with nodes representing available streams. Every stream has a pointer to one of its descendants. This pointer points to the root of a subtree of unused streams. The *spawn_rng* function updates all the spawning information in both the *struct* used to call the function and the new ones created. The different generators use different methods for realizing the tree paradigm, and we will flesh out the details below.

We first examine the main dichotomy among spawning methods for different generators. The Fibonacci generators force a binary tree structure upon all possible streams with each *struct* containing a spawn pointer to an unused stream. A call to *spawn_rng* will return streams at the top of this subtree, after which the original spawn pointer and new pointers reference the unused children of those nodes representing the created streams. The next two diagrams show the before and after states of spawn pointers in the binary tree method. The lines represent the tree structure and the arrows represent the spawn pointers. Here, we show a *spawn_rng* call that returns only one stream. The node labeled 0 represents the stream the function is called upon, and the node labeled 1 represents the stream instantiated by the call.

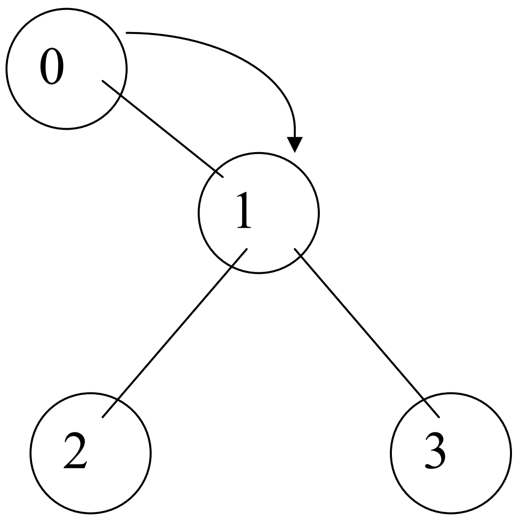

Figure 2.1 Binary tree spawning before spawn_rng is called on the node denoted by 0.

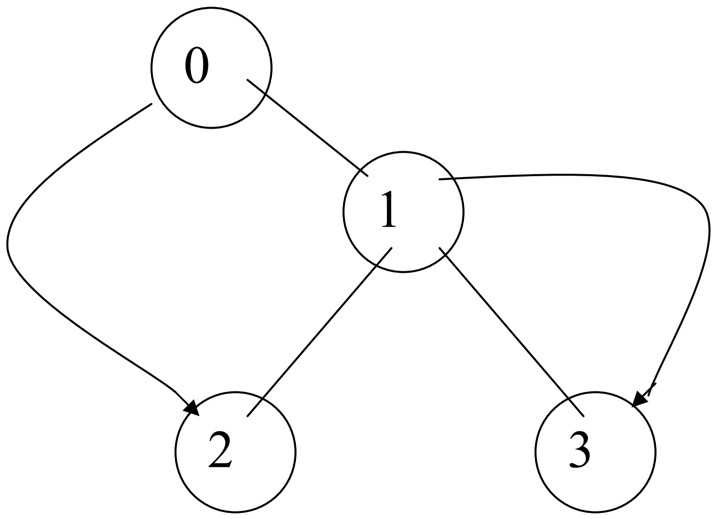

Figure 2.2 The binary spawn tree after spawn_rng is called on the node denoted by 0.

The generators with fewer streams available use a dynamic tree, where a node has as many children as the number of streams requested by the call to *spawn_rng* on that node. However, readers might find it more intuitive to think of this progression in a more linear fashion. Suppose we represent the j^{th} available stream with Y_j. A spawn pointer in this scheme is stored in terms of an offset, *k*, so that the spawn pool can be represented by:

$$\{Y_{j+nk} \mid n \geq 1 \text{ and } j + nk \leq M\},$$

where M is number of streams available in the generator. Whenever the *spawn_rng* function is called, requesting a number of streams, *r*, the new generators can be represented by:

$$Y_{j+k}, Y_{j+2k}, \ldots, Y_{j+rk}$$

Subsequent to the call, the new streams instantiated, along with the stream which was provided as input, have an offset of *rk*. As you can see, this method has a tree structure in its method of dividing spawn pools, though the structure of the tree is less obvious than that of the binary tree method.

The different spawning methods were created for the differing requirements of the various generators. The dynamic tree method seems more justified under the following logic when we consider the problem of "falling off the tree". Falling off the tree is the condition of having made a spawn call requesting more streams than in the spawn pool of the stream given as input. Not knowing *a priori* about which streams will be used in later spawning calls, the optimal condition following a spawn call is that all streams created by the call will have nearly equal spawn pools. This is a slight oversimplification; one might also posit that a stream that has been used in a spawn call will be more likely to spawn again in the future. The binary tree method accounts for this in the case of multiple streams being generated, and gives half of the remaining spawn pool to the stream used as input to a spawn call. This can cause problems if certain numbers of streams are instantiated at a time, but the main difference seems to lie in the division of spawn pools among the newly generated streams.

The dynamic tree method lets the spawn pools vary by at most one, where the binary tree allows some spawn pools to be twice as large as others. On the other hand, the binary tree method requires less memory. For the dynamic version, both the current stream number and the offset must be stored (stream numbers can be quite large in some of the generators). So for some generators, the dynamic method could require twice as much data as the binary version. Thus, the MLFG and LFG generators, which have the most streams and require the most memory, use the binary tree to reduce required memory. One should consider the particular generator for deciding which method is best. However, the current spawn functions in SPRNG have not been changed. The *init_rng* and *spawn_rng* functions have been left as is, and we have created a new method for instantiating streams that allows random access to all possible streams. We left the old *init_rng* and *spawn_rng* functions because of the usefulness they exhibit for a certain class of applications. The new method will be more general in nature.

Simulations cover a wide range of applications and produce much of the demand for random numbers. One of the most popular uses of the SPRNG library is particle simulation. In fact, we will see that the current SPRNG library has particle simulation at its heart in the design of initialization and spawning. Most particle simulations do well to have one random number stream per particle. The current method of spawning allows a user to create more streams from an already existing stream. Take for example the problem of neutron transport in nuclear reactions. The neutrons travel through some medium, and it has been observed that their motion is more appropriately described by probability distributions than a deterministic path. Random numbers are used to simulate such stochastic motion. The neutrons also cause events in the reaction. Sometimes they are absorbed, but sometimes they collide with other atoms causing fission. This fission creates more free neutrons that require a random number stream to determine their activity. This exemplifies the design of the *init_rng* and *spawn_rng* functions. They are designed to provide reproducibility in an environment where processes are branching according to the random numbers that have already been consumed. The *init_rng* function is used to provide enough streams to start the simulation, but the *spawn_rng* function is used afterwards whenever splitting requires new streams to be instantiated.

This is an ingenious design which prevents the user from having to know which streams have been consumed in order to instantiate new ones, but this method may feel restrictive in other situations. While reproducibility is an important concept, we believe many users would benefit from more control over the random number instantiation process, even if a little more calculation is required to maintain reproducibility. It becomes unnatural to use the initialization and spawning methods where particles are not created by splitting. Even when splitting is required, we feel many users would prefer to create their own methods of collision free numbering schemes to maintain reproducibility while saving some memory. We will now cover a few of the shortcomings of the current instantiation method.

As discussed earlier, many users have a problem with “falling off the tree”. In such a case, the user has not actually exhausted all the streams available; the user has simply tried to spawn from a node too close to the bottom of the tree to be able to correctly assign spawn pointers that would point to unused streams. In such a case, the current version of SPRNG simply changes the seed and points these to nodes elsewhere in the tree that may or may not be independent from those already been used. An error is given to the user warning that the streams in his application might no longer be independent of one another (in actuality, he might just be using the same streams again). For the most part, users have no idea how to use the spawn function to better effect, because they do not know of the underlying tree structure. For example, it is usually more efficient to spawn a large number of streams at a single time. The spawn function takes care that the new streams initialized are spread evenly at the top of the subtree. Deducing this requires knowledge of the tree structure. The user does not have the knowledge required either to make spawn calls in a way to progress through the tree evenly, or to understand why it would be better to make calls requesting larger numbers of streams instead of smaller ones. We have received much feedback from baffled users concerning falling off the tree. Some users have even been trying to always spawn from the same stream, which causes pointers to run down the side of the tree and fall off having initialized only a logarithmic portion of the streams that could have been used.

Another problem caused by poorly informed users is the unnecessary use of interprocessor communication. The SPRNG documentation says to call the *init_rng* function once on each processor, in order that subsequent *spawn_rng* calls can be made on individual processors without having to communicate stream information between processors. The tree methods are elegantly designed to avoid having to send this information between processors, but users do not seem to be aware of this. Some SPRNG users seem to think they should initialize all their streams on only one processor and then send the information to other processors. This again is probably due to a lack of knowledge of the underlying SPRNG architecture. When we see problems that arise because of poor use of the library, we have to conclude that the SPRNG interface is much less than intuitive for users in many situations. We cannot ask users to learn all about spawning trees so that they will use them better. The interface must be modified so that better information hiding can be accomplished.

A more tangible problem with this implementation, over which the user has no control, is the memory required in each stream. The LCGs do not require much memory, so a couple of words do not seem to matter that much. But users who need more streams than the LCGs can provide (or think they do because they do not understand spawning) need to use the Fibonacci generators for their massive parallelism. As discussed earlier, the Fibonacci generators require a large array of values for initialization with as many elements as the size of the lag. Moreover, the information required to implement the spawning structure in the Fibonacci generators is another array only one element smaller than the size of the lag. As the lag grows large, some users are unable to instantiate as many streams as they need because of the memory required to do so. For such users, to reduce the memory requirements is to allow more streams to be instantiated. We will discuss this issue more quantitatively after we have seen how to remove the requirement for this array by shifting to a new instantiation method.

CHAPTER 3

A NEW METHOD FOR CREATING STREAMS

The new *init_by_number* function gives random access to streams in a SPRNG generator. It allows the user to have access to all the independent streams in a generator, with independence guaranteed simply by using different stream numbers each time. In the old paradigm, stream numbers are hidden from the user, making it virtually impossible to use every available stream. For instance, many particle simulations use one stream per particle. So long as the particles are already numbered, creating the associated streams with *init_by_number* becomes extremely easy. A user need only pick a generator with as many streams as the largest number of any particle in the simulation. This would work similarly with applications that allow numbering of components that require a random number generator. So long as an upper bound can be established that is smaller than the number of streams for some generator, the user need not fear falling off the spawn tree, as is the case when using *spawn_rng*.

In fact, the previous method of spawning is so opaque as to prevent most casual users from determining if their spawn calls will fall off the tree in any but the most obvious cases. When the user receives an error message saying that he has fallen off the tree, under the old method he would have little information about how to solve the problem. We have even received feedback from users who have managed to discover the worst case (spawning from one generator or always from the last spawned) thinking they have found a bug in the package because they rapidly encounter this error message.

To some degree, the *init_rng* function provided random access to streams if used exclusively without the *spawn_rng* function. Instead of using the *init_rng* function to initialize streams at the beginning of a program, it could be used throughout. This functionality is not described in the users manual, because it is not the intended use of the *init_rng* function. This function also has a natural limit on the number of streams available through its use, because the stream number is passed as a single word. Many users need more streams than can be represented by one word. To complicate the problem, the *init_rng* function takes a signed integer as the stream number (required for the Fortran interface), but the actual parameter is required to be nonnegative, thus halving the number of streams available to the *init_rng* function. The new *init_by_number* function allows one to initialize any stream available.

In order to make the stream number parameter multi-precise and available to the user for manipulation, a standard for multi-precise arithmetic and representation has to be implemented within. The previous version of SPRNG had circumstances where multi-precise data was stored in arrays of integers, specifically for the Fibonacci generators. However, this internal representation was very nonstandard and was an impediment to user manipulation. For example, the library has internal functions for common operations to these arrays such as doubling and addition, but they by no means form a complete basis for multi-precise arithmetic, since only a few operations are necessary. For this reason, we have elected to use the *Gnu* multi-precise library (GMP) [12] for multi-precise parameters. While the formal arguments of the *init_by_number* function are pointers to native arrays, a conversion function is provided for GMP integers to the native types. In this way, the library is made to compile without the presence of GMP, but users who wish to use the *init_by_number* function instead of the old spawning routines are strongly advised to make use of GMP. The GMP library provides functions enabling these conversions.

The prototype for the new initialization function is:

*int init_by_number (int rng_type, unsigned *start, int nspawned, int ***newgens,*
int seed, int parameter, int checkid = 0);

The variables *nspawned, newgens*, and *checkid* carry the same meaning as in the *spawn_rng* function. The variable *nspawned* is the number of stream to be created. The variable *newgens* is a pointer which returns the newly created generators. It should point to an array of *int***, which has been allocated with an *int*** per stream to be created (*nspawned*). The *checkid* argument should be 0 for most runs, which is the default. The variables *rng_type, start, seed*, and *parameter* have all been used to replace information that was formerly pulled out of the stream *struct* that was passed to the *spawn_rng* function. The *rng_type* argument is the type of generator to be used. The variable *seed* is the value by which the user can get different random numbers for different runs of a program. The seed serves as the initial condition to each stream. The variable *parameter* chooses the parameterization scheme and should be adjusted according to memory and stream needs. The *start* argument takes a pointer to an array of the appropriate size and type for the particular *rng_type* and *parameter* chosen. This format can be achieved with the new *gmp2sn* function:

```
int* gmp2sn(int rng_type, int param, mpz_t stream_number );
```

This function takes a single GMP number, *stream_number*, as a parameter and returns a pointer allocated with the appropriate size and type of memory which is filled with the number given.

Note, detailed function descriptions of the new SPRNG implementation can be found in the appendix.

CHAPTER 4

BENEFITS OF THE NEW INSTANTIATION METHOD

We find that by allowing a user to access streams by stream number, benefits may be achieved over the old spawning method. Allowing a user random access to streams allows all possible streams to be exhausted, so long as the user is willing to do the bookkeeping. A user may also spawn on a particular processor without packing and unpacking the *struct* for the stream, as many of our uses have chosen to do with the old library. For those that choose to use *init_by_number* instead of spawning, the memory required to describe a stream can be reduced. While the changes made certainly increase functionality and efficiency, an equally important aspect is the simplicity of the interface the new functions provide. Although this function might require extra bookkeeping for some applications, some users might find this function definition easier to understand than that of the old method. All the user needs to know is that the same actual parameters to the function will yield the same sequence of numbers on any platform.

4.1 Exhausting all Streams

The previously mentioned problem of falling off the tree is very easy to avoid with the new *init_by_number* function. Under the old method, it is not hard for the user to fall off the tree without even exhausting the majority of streams. In the worst cases (either spawning from one stream repeatedly or spawning stream one at a time always from the most recently spawned), the spawn procedure fails after having only used *lg n* streams, where *n* streams are available.

4.2 Simplicity of Interface

Much of the misuse of the *init_rng* and *spawn_rng* functions will be assuaged by the simplicity of the interface of the new spawning method. Feedback indicates that many of our users are unaware of the proper usage of these two functions. In fact, most SPRNG users never call *spawn_rng.* Users might be confused about the dual function method of stream instantiation with difficulty recognizing which should be used for particular circumstances. While the documentation explains that the *init_rng* function can be called on different processors, feedback from users expresses their discomfort in doing so. The *pack* and *unpack* functions are used to store the data from a stream into a character array and rebuild the stream from that array after it has been sent as a message between processors. This means that sending a stream between processors requires both the overhead of the *pack* and *unpack* functions and a communication cost. Since sending streams between processors is expensive, our more advanced users would like to avoid doing so. It now suffices for the user to know that the same function parameters will yield the same sequence on any platform that SPRNG has been successfully ported to. For each seed, generator, and parameterization choice, the user gains random access to each stream available and can access them in sequential order, or any other order he sees fit.

4.3 Memory Reduction

The memory required for the old spawning approach is staggering for some generators. While those with few streams are implemented easily, those generators with many streams have to store a large stream number as a pointer to the next node to be visited. For some high performance users, memory becomes the prohibitive factor to the number of streams that can be instantiated. For example, the largest storage requirement in MLFG is an array of size L (in 64-bit unsigned integers), which can grow very large with choice of parameter. The size of this array depends on the lag chosen, as discussed earlier. An array to store spawn information is used, which is only one element shorter than the seed array. Say L = 17, which is the smallest L in the public distribution of

SPRNG, then the spawn information array for MLFG requires (17-1)*8 + 4 = 132 bytes. If one were to then initialize a large number (some use billions) of streams, this would create an enormous memory savings. The largest MLFG and LFG lags currently available are L = 1279. The default lags are different for MLFG and LFG. MLFG uses the smallest, 17, and LFG uses the largest, 1279. With the large lag, the spawn array requires over 8 kilobytes (kB) for an MLFG stream and 4 kB for an LFG stream. As the lag grows large the size of the eliminated array approaches half the size of the MLFG footprint and a third the size of the LFG footprint. For a user that is running out memory, this will allow twice as many MLFG streams to be initialized. When the lag is large, it does not take many stream initializations to receive a large benefit from not storing certain spawn information.

The user is afforded an opportunity for indirect memory savings as well. Since the new initialization method allows for exhaustion of all possible streams, the user can choose a generator/parameter with fewer possible streams to meet his needs. Generators with fewer possible streams consume less memory. This trade off between memory and number of streams available has frustrated users trying to accomplish very large, parallel applications. Making this tradeoff easier to deal with will affect parallel applications at the highest order of performance. Reduced memory consumption is one of the most compelling aspects of random access to streams.

CHAPTER 5

TESTING

Random number generators must be passed through a veritable gauntlet of statistical and physically inspired tests. These include testing streams for poor randomness within themselves and also for correlation among the many streams produced by a given generator and parameterization. However, these tests have already been run on the generators within SPRNG, and with the exception of well known faults in specific generators, SPRNG performs very well on these tests [14]. Thus, the aim of testing the *init_by_number* function is to insure that the same streams are instantiated as with the *init_rng* and *spawn_rng* functions. A random access numbering scheme has already been achieved that allows for the aforementioned functions, the random access was simply not fully available to the user. The general idea of the test procedure is simply to instantiate streams the old and new ways, and use them to check the first few numbers for consistency. Two approaches are the following.

The first is to call the *init_rng* and *spawn_rng* functions in a particular way and to calculate the respective stream numbers of each; then the stream numbers can be used as input for the *init_by_number* function and tested against each other. This procedure essentially insures stream number consistency between the old and new methods. This method is troublesome however; to test the streams to the extent that one would like for evidence that all components are working correctly requires fairly complicated calculations. One would hope to at least use the multi-precise aspects of the internal arrays to make sure that all the bits are in their proper place. Also, the testing procedure should be easier to manipulate than this.

Thus, an internal testing function was created, *get_stream_number*. While this function is not intended for users, it might be of help to users in the future for certain aspects of stream instantiation. This function is required since the stream number is not stored explicitly in some of the generators. Like the *init_by_number* function, it had to be crafted to each particular generator, and is probably responsible for just as many problems during debugging as the code being tested. In the generators that use the binary tree spawning method, the stream number must be recovered through the properties of the binary tree. The only time a spawn pointer points to the right child of a node is immediately after its instantiation. Afterwards, the pointer will move to the left child after being used as a parameter for the spawn function. One keeps moving up the tree until a right hand child is discovered. The underlying operation is to right shift the entire multi-precise array until a 1 is shifted off the least significant bit of the least significant element of the array. This, of course, must be platform independent, and tools are provided in SPRNG to make it so. With the aid of *get_stream_number*, any sequence of *init_rng* and *spawn_rng* function calls can be tested against the *init_by_number* versions of the proper stream. Testing the GMP conversion functions is rather straight forward, just making sure the right numbers are computed.

As for efficiency, there is no significant difference in resources required to instantiate streams between the old and new methods. The *init_by_number* function uses the same internal scheme as the old spawn functions. The only difference is that the stream number is handed through a function parameter instead of being pulled from a stream defining *struct* passed as a parameter. There are some minor speed savings given that spawn pointers are not being calculated. The *get_stream_number* function is not a user function, so it need not be tested for efficiency.

CHAPTER 6

MEMORY OPTIMIZATIONS TO SPRNG

Another goal in the development of SPRNG version 3.0 (SPRNG 3) is to reduce memory usage. Memory usage is an important issue for many SPRNG users. While the new stream instantiation method provides significant savings in the larger generators, efforts have been made to reduce memory usage for all users, not just those who choose to adopt the *init_by_number* function. Many opportunities for memory savings are available in SPRNG version 2.0 (SPRNG 2), and we have chosen to effect memory usage reduction through mostly minor changes.

6.1 The Name as a String

In SPRNG 2, a character pointer was assigned a string literal during initialization. The string was removed with the intention of saving the memory required to store the string. However, upon reevaluation it was realized that string literals are allocated statically, so after initialization the pointer was unstable. The pointer is still removed, but the only memory saved is that of the pointer itself. This change is more correctly viewed as removal of a wild pointer.

6.2 Generator Specific Optimizations

While memory optimization is always a concern in SPRNG, this section describes an effort to create a particularly small footprint for the multiplicative lagged-Fibonacci generator (MLFG). We use footprint to describe the memory consumed by each random number stream. Our goal was to create a generator with 50 bytes or less of memory. We miss this mark, but we do improve greatly over the smallest footprint that was available in the SPRNG 2 MLFG generator, which required over 300 bytes per stream. Because of the similarity between LFG and MLFG, these optimizations are applicable to LFG as well. Also required was the ability to spawn, so the work described in this paper that saves memory by removing the spawn ability could not be used.

The first optimization needed was to use a parameterization of the MLFG that lends itself to a small footprint. The MLFG generator uses a recursion based on two prior values determined by parameters *j* and *k*. We will let *k* be the larger; then the footprint will necessarily include *k* old values. The lowest value of *k* in any parameterization of MLFG in SPRNG 2 was 17. Values stored are 8 bytes in length. So with this parameterization, the bare requirements put the footprint over 100 bytes. We chose to include a new parameterization in which *k* is three. This provided the most significant memory reduction of any changes made. This parameterization of MLFG will be available in SPRNG 3, but the other parameterizations will be improved as well because of general changes to the implementation. Many of the changes in fact only reduce the footprint by a constant amount, rather than an amount dependent on *k*, so they do not provide savings on the scale that removing the spawning information was able to.

The struct used in MLFG is:

```
struct rngen
{
  int rng_type;  particular random generator to be used
  char *gentype;  name of the generator used
  int stream_number;  least significant bytes of the stream number
  int nstreams;
  int init_seed;  seed given by user before being XORed with the global seed
  int parameter;  determines the parameterization scheme
  int narrays;  variable used for packing and unpacking
  int *array_sizes;  variable used for packing and unpacking
  int **arrays;  variable used for packing and unpacking
  uint64 *lags;  internal array for producing random numbers
  uint64 *si; spawn pointer
  int hptr;          /* integer pointer into fill */
  int lval, kval, seed;  variables to store properties of parameterization scheme and seed
};
```

A few of the *int* type variables in this *struct* can be removed without impact on the functionality of the generator. However, as mentioned before, with the most common uses of the library considered, cutting out these values does not have much effect relative to the sizes of the streams generated. The l*val* and *kval* data can both be read by accessing into the array of suitable parameters, which lies in global memory, with the variable *parameter*. This requires an extra memory access per function call if it is to be stored in a local variable for the scope of the function. The variable *narrays* can be replaced by the constant literal 2.

The pointers require a little more explanation. As already mentioned, *gentype* is an unstable pointer that can be removed with only positive impact on the library. The *si* field is a pointer to the array of 64 bit unsigned integers that determines the next stream to be spawned if the spawn function is called on this stream. *lags* is a pointer to an array of similar size and type that serves to store the internal workings of the stream defined by the particular instantiation of the *struct*. These are the essentials of the stream and cannot be reduced or removed. However, *array_sizes* and *arrays* are not necessary to the *struct*. They can be replaced by local variables in the *pack* and *unpack* functions, which are the only functions to make use of these data. *arrays* is allocated memory of the size of 2 *int** which in turn alias the *lags* and *si* pointers. *array_sizes* is allocated memory of 2 *int* types, which store the size of the arrays. Since the pack and unpack functions are used so seldom (many times no more than once per stream instantiated), it makes much more sense to use local pointers to accomplish these purposes.

CHAPTER 7

CONCLUSIONS

We have developed a new stream instantiation method for the SPRNG library, embodied in the *init_by_number* function. While the old method provided some of this functionality already, many users will benefit from such a unified approach at exhaustive random access. Some users have not been using the old instantiation to full effect, mostly because of misunderstandings of the SPRNG documentation. In fact, most users never call *spawn_rng*. We complete the necessary work to make it a legitimate part of a well established and widely used mathematical library. This new method provides a more general level of access to the streams for those advanced users who feel rather handicapped by the old spawning method. While the old spawning method provides excellent access to random number streams in applications like random walks with splitting, we hope that this new instantiation will make SPRNG the library of choice for a broader range of random number needs. We have also performed memory optimization on the underlying data structure for streams to provide the smallest footprint possible.

With the added benefit of memory requirement reduction and communication requirement removal, the new SPRNG offers a very good choice to a broad range of Monte Carlo applications. Random access is the most powerful and general stream instantiation method that could be provided to users. Even the requirement for reproducibility is shifted to the user, allowing for special circumstances while preserving reproducibility as an easy property for the user to achieve. These changes should indeed broaden the base of SPRNG users and provide loyal users new tools for their applications. SPRNG is one of the most well researched and designed mathematical libraries available today and broadening its functionality is bound to be of use to many scientists.

CHAPTER 8

FUTURE WORK

Certainly, the improvements of user functionality and memory usage studied in this document could be extended. Also, some users prefer the SPRNG 1 library simply because of its architecture, where each generator could be compiled separately. Some users probably found this easier to manage, and most know which generator they need before use. The unified library compilation provides the ability to change generators within code by modifying only a function parameter. This is useful to those who wish to test different generators, providing a very easy change to those who find they have run out of streams or exhausted the period of some of their streams.

An obvious way to improve functionality is always to add new generators. Code for some new generators is currently available for future versions of SPRNG, but some minor flaws with them may still need to be mended. Because of the new instantiation method, new generators with a high number of streams like the MLFG and LFG can be added with full effect, and this is a very good way to improve the library. Also, it might be valuable to add a cryptographic random number generator to SPRNG. This is a generator designed to make it very difficult to predict the random numbers, even having seen many of the already generated numbers. One final addition might be to provide a SRPNG-like tool for parallel quasirandom number generation. While the SPRNG library has not been endowed with such functionalities yet, doing so would clearly widen the applications for which SPRNG would be an appropriate choice.

Another optimization that would be valuable to the library involves the generation of prime numbers. Some of the generators require each stream to use a different prime number as a parameter, in which case the i^{th} stream is provided with the i^{th} prime. The method for doing so in SPRNG is very likely sub optimal, causing a large slowdown in the time required to initialize a stream. The algorithm finds primes sequentially beginning with a prime stored in a file. These primes are sequential for the first several hundred, but for larger primes, the file stores only every 1000th prime. A simple solution is to store more primes in the file. Because of modern advances in data storage and communication, increasing the file size might be justified for the linear payoff in initialization speed it would provide.

APPENDIX A

SPRNG 3 USER'S GUIDE

SPRNG 3 is backwards compatible with SPRNG 2. This document describes new functionality in the SPRNG 3 library and is meant to supplement the existing user guide.

A.1 Installation

Dr. Yaohang Li added GNU autoconfig/automake scripts [6],[5] for ease of installation. To install the library, one should download the library and unzip/untar it to a directory of his choosing. At the top level is the SPRNG directory. In this directory, the user should call config, which creates a makefile according to the platform. The user can then use the make command to compile the library. This will create libsprng.a in the sprng/lib/ directory. This file should be linked to the user's program to compile.

A.2 Stream Instantiation.

A new method of instantiating random number streams has been added. The old spawning method, which uses the *init_rng* and *spawn_rng* functions, is still provided, and the user should not use the new instantiation method in the same program as in which these functions are used. The new function *init_by_number*, does not provide spawning information to the streams, so a call of *spawn_rng* using a stream created *init_by_number* will cause an error.

The *init_by_number* function is intended for advanced users who need millions or billions of streams. Not only does it allow exhaustion of all possible streams for a particular generator, parameter, and seed, it does not allocate the memory previously used to store spawning information for a significant reduction of memory usage per stream.

The *init_by_number* function is similar in its arguments to the *spawn_rng* function. It requires the user to keep track of the seed, parameter, and generator to take them as arguments. We recommend that a user use the same values of seed, parameter, and generator for each call. The user must also specify the stream number to be instantiated and the number of streams to be created, beginning at the specified stream number and produced sequentially thereafter. Thus, a user may create streams in blocks of any size that can be passed as an integer (while the stream number parameter provides for multi-precision, the number of streams parameter does not). The user then only needs to add the block size requested to arrive at the stream number to be passed to the next call of *init_by_number*. The user need not be concerned that the streams are being called sequentially; the seed will serve to create different numbers for different runs of a program. Many of the libraries simply permute the streams based on the seed. The generator and parameter are usually changed to affect the qualitative nature of the numbers (the generator) or the period and availability of the streams (parameters). These choices will be outlined more fully, including a couple of new parameters that have been added for improved versatility. The user may choose the generator and parameter based on the application, but once one is found that works, the user need only change the seed per run to produce different results.

Under SPRNG 2, the user was to call the *init_rng* function at the beginning of a program once per stream to be initially created. Each call took the number of times this function would be called in total as a parameter to provide the created stream with spawning information. Subsequently, any new streams to be instantiated required a call to the spawn function, which in turn required an existing stream. This method, while useful in many applications, has some drawbacks that will be corrected with a new instantiation method. Since the older spawn methods are still available, users must decide which method to use. THE METHODS SHOULD NOT BE MIXED. That is, the user must choose to use the *init_rng* and *spawn_rng* functions or the *init_by_number* function. Under no circumstances should the *spawn_rng* function be called with a stream that has been created by the *init_by_number* function. This follows from the *init_by_number* function's not storing spawning information in order to provide significant memory savings. The responsibility for independence of streams now rests with the user. However, the only requirement for streams to be independent, so long as the same generator, seed, and parameter are used for each call, is that each stream receives a different number for initialization. The user should keep in mind that if the *init_by_number* function is called to produce more than one stream, each stream receives numbers beginning sequentially at the stream number given as a parameter. This is the most important aspect of the new functions for users to understand and the reason it is recommended only for advanced users who need many streams. The user should also have GMP installed on the machine and enough basic arithmetic skill with GMP to produce the numbers required. Using the *init_by_number* function without GMP is very difficult, requiring the user to provide arrays of a certain size with the correct number embedded.

Some of the benefits to using the method of stream instantiation are reduced memory usage, no required communication cost for parallel applications, and the possibility of complete exhaustion of streams available. Users who just need a few streams should probably not use this method. LFG and MLFG are the generators benefiting the most from memory savings. Since these generators provide more streams than the others, removing the spawning information can provide very large memory savings (up to several kilobytes). Under the old method, many users with parallel applications were using interproccesor communication to distribute streams among processors. Since spawning requires an already existing stream, some processors had to produce streams to send to other processors. This not only makes for unbalanced work load, but the communication cost can be very high depending on the generator.

Users of the old spawning methods should be more careful to balance the calls to *init_rng* to prevent this, but users will find that the random access provides a more intuitive method of stream instantiation. While the *init_rng* function was to be called at the beginning of a program, with an argument specifying the number of times it is called, *init_by_number* can be used throughout the program and the only requirement is that the stream numbers be smaller than the largest stream number available. Finally, users who find they run out of streams quickly using the spawning routine (this typically generates an error message and returns a stream not guaranteed independent from the others) should use this method since it allows exhaustion of all possible streams. When the user receives an error message under the old spawning method, he has probably used less than half the streams available. In the worst case, the error message can be received having instantiated only lg N of N possible independent streams, where lg N is the logarithm to the base 2 of N.

The first item of business for those that wish to use the new instantiation method is to create a collision free numbering scheme to ensure properly independent streams. Streams can be initialized on any processor and the user might wish to designate a particular set of numbers, such as congruence classes modulus the number of processors, for each processor to use to prevent collision. Many applications will already be naturally numbered, such as a particle simulation. If one wishes to give a stream to each particle for example, so long as the particles are numbered uniquely, this number can be used for calls to the *init_by_number* function. One should also have an idea of the maximum number of streams to be used. If this number is such that it could be passed as a parameter to the *init_rng* function (the upper bound can be stored in an int type) the user may wish to simply use the *init_rng* function without the spawn function. Although not documented, the user can simply use the upper bound as the total number of calls to the *init_rng* function and can use these calls, once again using a unique number for each call, on different processors in a homogeneous cluster. However, the spawning information will still be stored by the streams, so memory might well be an issue that would convince a user to use *init_by_number* even if billions of streams are not needed. This makeshift method of random access to streams still makes interprocessor communication unnecessary and provides a significant advantage to those hindered by the hard cap on scalability dictated by the function parameters.

The prototype for the new function is:

*int init_by_number (int rng_type, unsigned *start, int nspawned, int ***newgens,* *int seed, int parameter, int checkid = 0);*

This can be compared to the old *spawn_rng* function:

*int spawn_rng(int *igenptr, int nspawned, int ***newgens, int checkid);*

nspawned, newgens, and *checkid* carry the same meaning as in the *spawn_rng* function. *nspawned* is the number of stream to be created. *newgens* is a pointer by which to return the newly created generators. It should point to an array of *int***, which has been allocated with an *int*** per stream to be created (*nspawned*). *checkid* should be 0 for final runs which it carries by default. *checkid* is nonzero when used for debugging purposes and its use is described in the SPRNG 2 user's manual.

rng_type, start, seed, and *parameter* have all been used to replace information that was formerly pulled out of the stream *struct*, *igenptr*, that was passed to the *spawn_rng* function. *rng_type* is the type of generator to be used. Current valid values are 0-5 with the following meanings:

0. LFG
1. LCG
2. LCG64
3. CMRG
4. MLFG
5. PMLCG (conditionally compiled according to the users installed libraries)

Macros are provided as in previous versions: SPRNG_LFG, SPRNG_LCG, and so on. *seed* is the value by which the user can achieve different numbers for different runs of a program. The seed serves as the initial condition to each stream. For some generators, this serves only to permute the order of the streams. *parameter* changes the values for the parameterization scheme. *parameter* should be adjusted according to memory and stream needs. These offer a trade off in memory used and number of streams available for instantiation. They affect periodicity as well. It is strongly recommended that the user use the same value of *rng_type*, *seed*, and *parameter* for each call in the program. *seed* should be changed between runs once the *rng_type* and *parameter* have been selected such that enough streams are available. If the user uses different seeds in the same run, he is in danger of using the same stream more than once, since the seed is used as a bitmask to permute the streams in some generators. The *start* argument takes a pointer to an array of the appropriate size and type for the particular *rng_type* and *parameter* chosen. This format can be achieved with the *gmp2sn* function:

int gmp2sn(int rng_type, int param, mpz_t stream_number);*

This function takes a single GMP number, *stream_number* as a parameter and returns a pointer allocated with the appropriate size and type of memory which is filled with the number given. The programmer should be careful to free the memory returned by this function, so it is important to assign it to a pointer before passing it as an argument. The final product should look something like:

int stream_number = gmp2sn(rng,param,stream);* where *stream* is a GMP integer
init_by_number(rng_type, stream_number, ...);
free(stream_number);

Users of PMLCG should also call *mpz_clear* before freeing the returned pointer, since the returned value is a GMP integer. The user should also take care to handle the freeing of the GMP numbers upon outliving their use for generating streams.

A.3 Adding generators

The SPRNG library is designed to be expandable not only by those who maintain the code for centralized distribution, but also for users who want to implement their own generator with the SPRNG architecture. The format in doing so has changed slightly in that the programmer must add the *init_by_number* function to the general format of a generator. This should not be that hard to do, as a prerequisite for a SPRNG generator is the ability to produce streams in this fashion. We recommend that the *init_by_number* be used as a sort of backend to the *init_rng* and *spawn_rng* functions. Most of the generators already had a similar function that served these functions with the internals of the stream. As before, the *init_rng* and *spawn_rng* functions should be responsible for processing the spawning information so that they will function as desired. A GMP conversion function should also be provided if the programmer requires this capability.

APPENDIX B

SPRNG 3 CODE ADDITIONS

B.1 LCG functions

```
/*********************************************************************
    init_by_number  allows users to retrieve by stream numbers.
    recommended for advanced users only.  Added by Jason Parker
    as part of sprng 3.0
*********************************************************************/

#ifdef __STDC__
int init_by_number(int rng_type, unsigned *start,  int nspawned, int ***newgens, int
seed, int mult, int checkid)
#else
int init_by_number(rng_type, start,nspawned, newgens, seed, mult, checkid)
int rng_type, nspawned, ***newgens, seed, mult, checkid;
unsigned* start;
#endif
{
  struct rngen **genptr;
  int i, j,tmult;
  tmult = mult;
  if (nspawned <= 0) /* check if nspawned is valid */
  {
    nspawned = 1;
    errprint("WARNING","spawn_by_number","nspawned <= 0. Default value of 1 used
for nspawned");
  }
  if (mult < 0 || mult >= NPARAMS)
  {
    errprint("WARNING","spawn_by_number","multiplier not valid. Using Default
param");
    tmult = 0;
  }

  genptr = (struct rngen **) mymalloc(nspawned*sizeof(struct rngen *));
  if(genptr == NULL)
  {
    *newgens = NULL;
    return 0;
  }
  for(i=0; i<nspawned; i++)
  {
    genptr[i] = (struct rngen *) mymalloc(sizeof(struct rngen));
    if(genptr[i] == NULL)
    {
      nspawned = i;
      break;
    }

    genptr[i]->init_seed = seed;
    genptr[i]->prime_position = *start + i;
    if(genptr[i]->prime_position > MAXPRIMEOFFSET)
    {
      fprintf(stderr,"WARNING - spawn_rng: gennum: %d > maximum number of
independent streams: %d\n\tIndependence of streams cannot be guranteed.\n",
            genptr[i]->prime_position, MAX_STREAMS);
      genptr[i]->prime_position %= MAXPRIMEOFFSET;
    }

    genptr[i]->prime_next = genptr[i]->prime_position + nspawned;
    getprime_32(1, &(genptr[i]->prime), genptr[i]->prime_position);

    genptr[i]->parameter = tmult;



    genptr[i]->rng_type = rng_type;
    genptr[i]->gentype = GENTYPE;

#ifdef LONG64
  genptr[i]->seed = INIT_SEED;     /* initialize generator */
  genptr[i]->seed ^= ((unsigned LONG64) seed)<<16;
  genptr[i]->multiplier = mults[tmult];
  if (genptr[i]->prime == 0)
    genptr[i]->seed |= 1;
#else
  genptr[i]->seed[1] = 16651885^((seed<<16)&(0xff0000));/* initialize generator */
  genptr[i]->seed[0] = 2868876^((seed>>8)&(0xffffff));
  genptr[i]->multiplier = mults[tmult];
  if (genptr[i]->prime == 0)
    genptr[i]->seed[1] |= 1;
#endif

    if(genptr[i]->prime_position > MAXPRIMEOFFSET)
      advance_seed(genptr[i]); /* advance lcg 10^6 steps from initial seed */

    for(j=0; j<LCGRUNUP*(genptr[i]->prime_position); j++)
```

```
     get_rn_dbl( (int *) genptr[i]);
  }

  NGENS += nspawned;

  *newgens = (int **) genptr;
  if(checkid != 0)
   for(i=0; i<nspawned; i++)
     if(addID(( int *) genptr[i]) == NULL)
          return i;

  return nspawned;
}

/*********************************************************************
  gmp2sn is used to convert a gmp number to the correct format for
  the init_by_number function.  Added by Jason Parker as a part of
  SPRNG 3.0.
*********************************************************************/

#ifdef USE_PMLCG
#ifdef __STDC__
int* gmp2sn(int rng_type, int param, mpz_t a)
#else
int* gmp2sn(rng_type, param, a)
int rng_type, param;
mpz_t a;
#endif
{
  int* b;
  b = (int *)malloc(sizeof(int));
  *b = mpz_get_si(a);
  if(mpz_cmp_si(a,*b)!=0)
    fprintf(stderr,
    "GMP Integer too large to be converted for the lcg generator\n");
  return b;
}
#endif
```

B.2 LCG64 functions

```
/*************************************************************
  init_by_number allows users random access to streams.  See
  documentation for more details.  Added by Jason Parker
  as part of SPRNG 3.0.
*************************************************************/


#ifdef __STDC__
int init_by_number(int rng_type, unsigned *start,  int nspawned, int ***newgens, int
seed, int mult, int checkid)
#else
int init_by_number(rng_type, start, nspawned, newgens, seed, mult, checkid)
int rng_type, nspawned, ***newgens, seed, mult, checkid;
unsigned* start;
#endif
{
  struct rngen **genptr;
  int i, j;
```

```
 if (nspawned <= 0) /* is nspawned valid ? */
 {
   nspawned = 1;
   fprintf(stderr,"WARNING - spawn_rng: nspawned <= 0. Default value of 1 used for
nspawned\n");
 }

 genptr = (struct rngen **) mymalloc(nspawned*sizeof(struct rngen *));
 if(genptr == NULL)    /* allocate memory for pointers to structures */
 {
   *newgens = NULL;
   return 0;
 }

 for(i=0; i<nspawned; i++)  /* create nspawned new streams */
 {
   int gennum;

   gennum = *start + i;

   if(gennum > MAX_STREAMS)   /* change seed to avoid repeating sequence */
     seed = seed^gennum;
   else
     seed = seed;
   /* Initialize a stream. This stream has incorrect spawning information.
      But we will correct it below. */

   genptr[i] = (struct rngen *)
     init_rng(rng_type,gennum, gennum+1, seed, mult);
```

```
    if(genptr[i] == NULL)       /* Was generator initiallized? */
    {
      nspawned = i;
      break;
    }
    genptr[i]->spawn_offset = (nspawned+1);
  }





  *newgens = (int **) genptr;



  if(checkid != 0)
   for(i=0; i<nspawned; i++)
     if(addID(( int *) genptr[i]) == NULL)
          return i;

  return nspawned;

}

/****************************************************************
  gmp2sn is a user function that allows conversion of a GMP number
  to the appropriate format for the init_by_number function.
  Added by Jason Parker as part of SPRNG 3.0
****************************************************************/


#ifdef USE_PMLCG
```

```
#ifdef __STDC__
int* gmp2sn(int rng_type, int param, mpz_t a)
#else
int* gmp2sn(rng_type, param, a)
int rng_type, param;
mpz_t a;
#endif
{
  int * b;
  b = (int *)malloc(sizeof(int));
  *b = mpz_get_ui(a);
  if(mpz_cmp_si(a,*b)!=0)
    fprintf(stderr, "GMP integer too big for lcg64 generator\n.");
  return b;
}
#endif
```

B.3 CMRG functions

```
/*****************************************************************
  init_by_number allows users random access to streams.  See
  documentation for additional details.  Added by Jason Parker
  as part of SPRNG 3.0
*****************************************************************/

#ifdef __STDC__
int init_by_number(int rng_type, unsigned *start, int nspawned, int ***newgens,int seed,
int param, int checkid)
#else
int init_by_number(rng_type,start,nspawned, newgens, seed, param, checkid)
int rng_type,nspawned, ***newgens, seed, param,checkid;
```

```
unsigned* start;
#endif
{
  struct rngen **genptr;
  int i, j;

  if (nspawned <= 0) /* is nspawned valid ? */
  {
    nspawned = 1;
    fprintf(stderr,"WARNING - spawn_rng: nspawned <= 0. Default value of 1 used for
nspawned\n");
  }

  genptr = (struct rngen **) mymalloc(nspawned*sizeof(struct rngen *));
  if(genptr == NULL)     /* allocate memory for pointers to structures */
  {
    *newgens = NULL;
    return 0;
  }

  for(i=0; i<nspawned; i++)  /* create nspawned new streams */
  {



    /* Initialize a stream. This stream has incorrect spawning information.
       But we will correct it below. */

    genptr[i] = (struct rngen *)
      init_rng(rng_type,*start+i, *start+i+1, seed, param);
    if(genptr[i] == NULL)        /* Was generator initiallized? */
    {
      nspawned = i;
      break;
    }
    genptr[i]->spawn_offset = (nspawned+1);
  }

  /*tempptr->spawn_offset *= (nspawned+1);*/



  *newgens = (int **) genptr;



  if(checkid != 0)
   for(i=0; i<nspawned; i++)
     if(addID(( int *) genptr[i]) == NULL)
          return i;

  return nspawned;
}

/*************************************************************************
  gmp2sn is called by the user to convert a gmp number to the correct
  format for the init_by_number funciton.  Added by Jason Parker as a
  part of SPRNG 3.0
*************************************************************************/

#ifdef USE_PMLCG
#ifdef __STDC__
int* gmp2sn(int rng_type, int param, mpz_t a)
```

```
#else
int* gmp2sn(rng_type, param, a)
int rng_type, param;
mpz_t a;
#endif
{
  int * b;
  b = (int *)malloc(sizeof(int));
  *b = mpz_get_ui(a);
  if(mpz_cmp_si(a,*b)!=0)
    fprintf(stderr,"GMP integer too large for cmrg generator");
  return b;
}
#endif
```

B.4 LFG functions

```
/*****************************************************************
  init_by_number allows users random access to streams.  See
  documentation for details.  Added by Jason Parker as part of
  SPRNG 3.0
*****************************************************************/

#ifdef __STDC__
int init_by_number(int rng_type, unsigned *start,  int nspawned, int ***newgens, int
seed, int param, int checkid)
#else
int init_by_number(rng_type, start,nspawned,newgens,seed,param, checkid)
int rng_type, nspawned, ***newgens, seed, param, checkid;
unsigned* start;
#endif
```

```
{
  int **q=NULL, i;
  unsigned *p;


  if (nspawned <= 0) /* check if nspawned is valid */
  {
    nspawned = 1;
    errprint("WARNING","spawn_rng","Nspawned <= 0. Default value of 1 used for
nspawned");
  }



  p = start;
  q = initialize_by_number(rng_type,nspawned,param,seed^GS0,p,seed);
  if (q == NULL)
    {
      *newgens = NULL;
      return 0;
    }

  NGENS += nspawned;

  *newgens = (int **) q;

  if(checkid != 0)
   for(i=0; i<nspawned; i++)
     if(addID((*newgens)[i]) == NULL)
          return i;
  return nspawned;
}

/*****************************************************************
  initialize_by_number is called by the init_by_number funtion.  It
  is not a user function.  Added by Jason Parker as part of SPRNG 3.0
*****************************************************************/

#ifdef __STDC__
static int **initialize_by_number(int rng_type, int ngen, int param, unsigned seed,
unsigned *nstart, unsigned initseed)
#else
static int **initialize_by_number(rng_type, ngen,param, seed,nstart, initseed)
int rng_type,  ngen, param;
unsigned *nstart, seed, initseed;
#endif
{
  int i,j,k,l, length,run,temp;
  struct rngen **q;
  unsigned *nindex1,*nindex2;

  length = valid[param].L;

/*      allocate memory for node number and fill of each generator      */
  temp = nstart[0];
  q = (struct rngen **) mymalloc(ngen*sizeof(struct rngen *));
  if (q == NULL)
    return NULL;
 for (i=0;i<ngen;i++)
 {
   q[i] = (struct rngen *) mymalloc(sizeof(struct rngen));
```

```
    if (q[i] == NULL)
      return NULL;

    q[i]->rng_type = rng_type;
    q[i]->hptr = length - 1;
    q[i]->si = NULL;
    q[i]->r0 = (unsigned *) mymalloc(length*sizeof(unsigned));
    q[i]->r1 = (unsigned *) mymalloc(length*sizeof(unsigned));
/*    q[i]->lval = length;
    q[i]->kval = valid[param].K;*/
    q[i]->param = param;
    q[i]->seed = seed;
    q[i]->init_seed = initseed;

    if (q[i]->r1 == NULL || q[i]->r0 == NULL)
      return NULL;
  }
/*      specify register fills and node number arrays                    */
/*      do fills in tree fashion so that all fills branch from index     */
/*           contained in nstart array                                   */
  q[0]->stream_number = nstart[0];
  nindex1 = (unsigned*)malloc((length-1)*sizeof(unsigned));
  nindex2 = (unsigned*)malloc((length-1)*sizeof(unsigned));
  for(i = 0; i < length-1; i++){
    nindex1[i]=nstart[i];
  }
  run = 0;
  for(i=1; i < length-1; i++)
    if(nindex1[i]!=0){
      run = 1;
      break;
```

```
 }

si_double(nindex2,nindex1,length);
get_fill(nindex2,q[0]->r0,param,seed);
nindex2[0]++;
get_fill(nindex2,q[0]->r1,param,seed);
for(i = 1; i < ngen; i++){
  si_add_one(nindex1,length-1);
  for (j=0;j<length-1;j++)
              nindex2[j] = nindex1[j];
  q[i]->stream_number = nindex1[0];
  si_double(nindex2,nindex1, length);

  get_fill(nindex2,q[i]->r0,param,seed);
  nindex2[0]++;
  get_fill(nindex2,q[i]->r1,param,seed);
}
i = 0;
if(run==0)
for (i=0;i<ngen;i++,temp++)
{
 /*
 k = 0;
 for (j=1;j<lval-1;j++)
   if (q[i]->si[j])
        k = 1;
 if (!k)
   break;
 */
 for (j=0;j<length*4;j++)
   get_rn_int((int *)(q[i]));
```

```
    if(temp==INT_MASK){
      i++;
      break;
    }
  }
  while (i<ngen)
  {
   for (j=0;j<RUNUP*length;j++)
      get_rn_int((int *)(q[i]));
    i++;
  }

  return((int **)q);
}

/*****************************************************************
  gmp2sn is used to convert gmp numbers to the correct format for
  the init_by_number function.  Added by Jason Parker as part of
  SPRNG 3.0.
*****************************************************************/

#ifdef USE_PMLCG
#ifdef __STDC__
int* gmp2sn(int rng_type, int param, mpz_t a)
#else
int* gmp2sn(rng_type, param, a)
int rng_type, param;
mpz_t a;
#endif
{
  int length, i;
  unsigned *ret;
  mpz_t b,c;
  mpz_init_set(b,a);
  mpz_init(c);
  length = valid[param].L -1;
  ret = (unsigned *)malloc(length*sizeof(unsigned));
  for(i = 0; i < length; i++)
    ret[i]=0;
  for(i = 0; i < length; i++){
    if(mpz_cmp_ui(b,0)==0)
      break;
    mpz_tdiv_r_2exp(c,b,MAX_BIT_INT+1);
    ret[i] = INT_MASK & mpz_get_ui(c);
    mpz_tdiv_q_2exp(b,b,MAX_BIT_INT+1);
  }
  mpz_clear(c);
  mpz_clear(b);
  return (int *)ret;
}
#endif
```

B.5 MLFG functions

```
/*****************************************************************
  init_by_number allows users random access to streams.  See
  documentation for details.  Added by Jason Parker as part of
  SPRNG 3.0.
*****************************************************************/

#ifdef __STDC__
```

```
int init_by_number(int rng_type, unsigned *start,  int nspawned, int ***newgens, int
seed, int mult, int checkid)
#else
int init_by_number(rng_type, start, nspawned, newgens, seed, mult, checkid)
int rng_type, nspawned, ***newgens, seed, mult, checkid;
unsigned* start;
#endif
{
            struct rngen **genptr;
  int i;
  uint64 *p;

  if (nspawned <= 0) /* is nspawned valid ? */
  {
    nspawned = 1;
    fprintf(stderr,"WARNING - spawn_rng: nspawned <= 0. Default value of 1 used for
nspawned\n");
  }

  p = (uint64*)start;
  seed &=0x7FFFFFFF;

  genptr = initialize_by_number(rng_type, nspawned,mult,seed^GS0,p,seed);
  if(genptr == NULL)    /* allocate memory for pointers to structures */
  {
    *newgens = NULL;
    return 0;
  }

  si_double(p,p,valid[mult].L);
/*
```

```
 for(i=0; i<nspawned; i++)
 {
   genptr[i]->array_sizes = (int *) mymalloc(genptr[i]->narrays*sizeof(int));
   genptr[i]->arrays = (int **) mymalloc(genptr[i]->narrays*sizeof(int *));
   if(genptr[i]->array_sizes == NULL || genptr[i]->arrays == NULL)
     return 0;
   genptr[i]->arrays[0] = (int *) genptr[i]->lags;
   genptr[i]->arrays[1] = (int *) genptr[i]->si;
   genptr[i]->array_sizes[0] = genptr[i]->lval*sizeof(uint64)/sizeof(int);
   genptr[i]->array_sizes[1] = (genptr[i]->lval-1)*sizeof(uint64)/sizeof(int);
 }
*/
  NGENS += nspawned;

  *newgens = (int **) genptr;

  if(checkid != 0)
   for(i=0; i<nspawned; i++)
     if(addID(( int *) genptr[i]) == NULL)
         return i;

  return nspawned;

}

/****************************************************************
  intialize_by_number is called by init_by_number.  It is not a
  user function.  Added by Jason Parker as part of SPRNG 3.0
****************************************************************/
```

```
static struct rngen **initialize_by_number(int rng_type, int ngen, int param, unsigned int
seed, uint64 *nstart, unsigned int initseed)
{
  int i,j,k,l,m,*order, length,run;
  struct rngen **q;
  uint64 *nindex, temp1, temp2, mask,mask2,t,r;
  set(nstart[0],t);
  length = valid[param].L;

  q = (struct rngen **) mymalloc(ngen*sizeof(struct rngen *));
  if (q == NULL)
    return NULL;

  for (i=0;i<ngen;i++)
  {
    q[i] = (struct rngen *) mymalloc(sizeof(struct rngen));
    if (q[i] == NULL)
      return NULL;

    q[i]->rng_type = rng_type;
    q[i]->hptr = 0;   /* This is reset to lval-1 before first iteration */
    q[i]->si = /*(uint64 *) mymalloc((length-1)*sizeof(uint64))*/ NULL;
    q[i]->lags = (uint64 *) mymalloc(length*sizeof(uint64));
   q[i]->parameter = param;
    q[i]->seed = seed;
    q[i]->init_seed = initseed;

    if (q[i]->lags == NULL/* || q[i]->si == NULL*/)
      return NULL;
  }
```

```
/*      specify register fills and node number arrays                    */
/*      do fills in tree fashion so that all fills branch from index     */
/*           contained in nstart array                                   */

  q[0]->stream_number = lowword(nstart[0]);
  get_fill(nstart,q[0]->lags,param,seed);
/*  si_double(q[0]->si,nstart,length);*/
  nindex = (uint64*)malloc((length-1)*sizeof(uint64));
  for(i =0; i < length-1; i++)
    set(nstart[i],nindex[i]);
  run = 0;
  for(i = 1; i < length-1; i++)
    if(notzero(nindex[i])){
      run = 1;
      break;
    }
  set(ONE,mask);
  for(m=0; m<length; m++)
  {
    and(SEED_MASK,mask,temp1);
    if(notzero(temp1))
      findseed(1,q[0]->lags[m], &q[0]->lags[m]);
    else
      findseed(0,q[0]->lags[m], &q[0]->lags[m]);
    lshift(mask,1,mask);
  }
  /*
  add(q[0]->si[0],ONE,q[0]->si[0]);

  i = 1;
  order[0] = 0;*/
```

```
lshift(ONE,MAX_BIT_INT+1,mask2);
if (ngen>1)
  for (i=1;i<ngen;i++)
  {
      for(j = 0; j < length-1; j++){
      add(nindex[j],ONE,nindex[j]);
      and(nindex[j],mask2,temp2);
      if(notzero(temp2)){
        xor(nindex[j],mask2,nindex[j]);
        if(j==length-2){
         fprintf(stderr,"ERROR: Invalid stream number used for\
               initialization, Independence of streams not gauranteed.");
        }
      }
      else{
       break;
      }
    }
      q[i]->stream_number = lowword(nindex[0]);
      get_fill(nindex,q[i]->lags,param,seed);
      set(ONE,mask);
      for(m=0; m<length; m++)
      {
        and(SEED_MASK,mask,temp1);
        if(notzero(temp1))
         findseed(1,q[i]->lags[m], &q[i]->lags[m]);
        else
         findseed(0,q[i]->lags[m], &q[i]->lags[m]);
        lshift(mask,1,mask);
      }
```

```
          if (ngen == i+1)
          break;
    }
   free(nindex);
  i=0;
  if(run==0)
  for (i=0;i<ngen;i++)
  {
   for (j=0;j<length*4;j++)
     advance_state(q[i]);
   xor(t,INT_MASK,r);
   if(!notzero(r)){
     i++;
     break;
   }
   add(t,ONE,t);
  }

  while (i<ngen)
  {
   for (j=0;j<RUNUP*length;j++)
     advance_state(q[i]);
   i++;
  }
  return (int**)q;
}
```

/**

gmp2sn converts a GMP number to the correct format for the *init_by_number* function. Added by Jason Parker as part of

```
  SPRNG 3.0
***************************************************************/

#ifdef USE_PMLCG
#ifdef __STDC__
int* gmp2sn(int rng_type, int param, mpz_t a)
#else
int* gmp2sn(rng_type, param, a)
int rng_type, param;
mpz_t a;
#endif
{
  int length, i;
  mpz_t b,c,d;
  uint64* ret;
  uint64 t;
  mpz_init_set(b,a);
  mpz_init(c);
  mpz_init(d);
  length = valid[param].L - 1;
  ret = (uint64*)malloc(length*sizeof(uint64));
  for(i = 0; i < length; i++)
    ret[i] = 0;
  for(i = 0; i < length; i++){
    if(mpz_cmp_ui(b,0)==0)
      break;
    mpz_tdiv_r_2exp(c,b,MAX_BIT_INT+1);
    mpz_tdiv_r_2exp(d,c,32);
    mpz_tdiv_q_2exp(c,c,32);
    ret[i] = mpz_get_ui(c);
    lshift(ret[i],32,ret[i]);
    t = mpz_get_ui(d);
    or(ret[i],t,ret[i]);
    mpz_tdiv_q_2exp(b,b,MAX_BIT_INT+1);
  }
  mpz_clear(b);
  mpz_clear(c);
  mpz_clear(d);
  return ret;
}
#endif
```

B.6 PMLCG functions

```
/*****************************************************************
  init_by_number gives users random access to streams.  See
  documentation for details.
  Added by Jason Parker as part of SPRNG 3.0
*****************************************************************/



#ifdef __STDC__
int init_by_number(int rng_type, unsigned* start, int nspawned, int ***newgens, int seed,
int mult, int checkid)
#else
int init_by_number(rng_type, start, nspawned, newgens, seed, mult, checkid)
int rng_type, nspawned, ***newgens, seed, mult, checkid;
unsigned* start;
#endif
{
        struct rngen **genptr;
  int i;
```

```
 if (nspawned <= 0) /* is nspawned valid ? */
 {
   nspawned = 1;
   fprintf(stderr,"WARNING - spawn_rng: nspawned <= 0. Default value of 1 used for
nspawned\n");
 }

 genptr = (struct rngen **)
   initialize_by_number(nspawned,(MP_INT*) start,seed,mult);
 if(genptr == NULL)     /* allocate memory for pointers to structures */
 {
   *newgens = NULL;
   return 0;
 }
 else
 {
   *newgens = (int **) genptr;
   for(i=0; i<nspawned; i++)
   {
     genptr[i]->rng_type = rng_type;
     genptr[i]->gentype = GENTYPE;
     genptr[i]->stream_number = *start + i;
     genptr[i]->nstreams = 0;
     genptr[i]->init_seed = seed;
     genptr[i]->parameter = mult;

     genptr[i]->narrays = 0;              /* number of arrays needed by your generator */

     NGENS++;
   }
 }
 if(checkid != 0)
  for(i=0; i<nspawned; i++)
    if(addID(( int *) genptr[i]) == NULL)
          return i;

 return nspawned;

}

/*********************************************************************
 initialize_by_number is called by the init_by_number function.
 It is not a user function.
 Added by Jason Parker as part of SPRNG 3.0
*********************************************************************/

int **initialize_by_number(int ngen, MP_INT *old_si, int seed, int param)
{
 /*
   called by: init_by_number
   calls    : init()
   GMP routines

   params   : int ngen = number of generators to initialize
   MP_INT old_si = value of k to use for first generator produced
   seed = encoding of starting state of generator
   param = power that determines Merssene prime
   returns  : pointer to pointers to RNGs (rngen structures)

   Initializes 'ngen' new generators
   ( allocates memory and gives initial values to the elements of 'rngen' )
```

```
    */
  int i,k,l,*order;
  struct rngen **q;
  static unsigned long a[2], r[2];
  int a_size;
  mpz_t temp;
  mpz_init(temp);
  order = (int *) mymalloc(ngen*sizeof(int));
  /* allocate memory for 'ngen' generators */
  q = (struct rngen **) malloc(ngen * sizeof(struct rngen *));
  if (q==NULL || order==NULL)
    return ((int **)NULL);
  for (i=0; i<ngen; i++)
  {
    q[i] = (struct rngen *) malloc(sizeof(struct rngen));
    if(q[i] == NULL)
      return NULL;

    mpz_init(&(q[i]->k));
  }

  /* set up 1st generator */
  mpz_set(&(q[0]->k),old_si);
#ifdef LONG64
  a_size = init(a, r, &(q[0]->k),seed,param);
  q[0]->mult = (unsigned LONG64)a[1]<<32|a[0];
  q[0]->x = (unsigned LONG64)r[1]<<32|r[0];
#else
  q[0]->a_size = init(q[0]->a, q[0]->r, &(q[0]->k),seed,param);
#endif
```

```
  mpz_set(temp, old_si);
  /* set up remaining generators */
  i = 1;
  order[0] = 0;
  if (ngen>1) while (1)
  {
     mpz_add_ui(temp,temp,1);
     mpz_set(&(q[i]->k), temp);
#ifdef LONG64
     a_size = init(a,r,&(q[i]->k),seed,param);
     q[i]->mult = (unsigned LONG64)a[1]<<32|a[0];
     q[i]->x = (unsigned LONG64)r[1]<<32|r[0];
#else
     q[i]->a_size = init(q[i]->a,q[i]->r,&(q[i]->k),seed,param);
#endif
     if (ngen == ++i)
          break;
  }
  mpz_clear(temp);
  free(order);
  return( (int **)q );

}

/*****************************************************************
  gmp2sn converst a gmp number to the correct format for the
  init_by_number function.  Added by Jason Parker as part of
  SPRNG 3.0
*****************************************************************/

int* gmp2sn(int rng_type, int param, mpz_t a){
  MP_INT *b;
  b = (MP_INT*)malloc(sizeof(MP_INT));
  mpz_init(b);
  mpz_set(b,a);
  return (int*)b;
}
```

BIBLIOGRAPHY

1. T. Bass. The Newtonian Casino. Penguin, London: 1990.

2. D. Burton. Elementary Number Theory. McGraw-Hill, New York: 1997.

3. A. De Matteis and S. Pagnutti, "Parallelization of random number generators and long-range correlations." Parallel Computing, **15**: 155-164, 1990.

4. A. De Matteis and S. Pagnutti, "Long-range correlations in linear and non-linear random number generators." Parallel Computing, **14**: 207-210, 1990.

5. Free Software Foundation. GNU Autoconf, http://www.gnu.org/software/autoconf/

6. Free Software Foundation. GNU Automake. http://www.gnu.org/software/automake/automake.html

7. Free Software Foundation. GNU MP, http://www.gnu.org/software/gmp/gmp.html

8. D. Knuth. The Art of Computer Programming vol. 2 Seminumerical Algorithms. 3rd ed. Addison-Wesley, Reading: 1998.

9. D. Lehmer. "Mathematical Method in Large-scale Computing Units." 2nd Symposium on Large-Scale Digital Calculating Machinery, 141-146, Harvard U. P., Cambridge: 1951.

10. G. Marsaglia. "Random Numbers Fall Mainly in the Planes." Proceedings of the National Academy of Sciences of the United States of America, 62: 25-28, 1968.

11. M. Mascagni and A. Srinivasan. "SPRNG: A Scalable Library for Pseudorandom Number Generation." ACM Transactions on Mathematical Software, **26**:436-461, 2000.

12. M. Mascagni. Personal Communication.

13. H. Niederreiter. Random Number Generation and Quasi-Monte Carlo Methods. SIAM, Philadelphia: 1992.

14. A. Srinivasan, M. Mascagni, and D. Ceperley. "Testing Parallel Random Number Generators". Parallel Computing, **29**:69-94, 2003.

BIOGRAPHICAL SKETCH

Jason Parker was born in Memphis, TN in 1979. He remained there and received a high school diploma from Memphis Harding Academy in 1997. He earned a Bachelor of Science in Mathematics and Computer Science from Florida State University in 2001.