SPRNG: A Scalable Library for Pseudorandom Number Generation

Michael Mascagni (mascagni@cs.fsu.edu)
Dept. of Computer Science, Florida State University
and
Ashok Srinivasan (ashok@math.iitb.ernet.in)
Dept. of Mathematics, Indian Institute of Technology–Bombay

In this article we present background, rationale, and a description of the **S**calable **P**arallel **R**andom **N**umber **G**enerators (SPRNG) library. We begin by presenting some methods for parallel pseudorandom number generation. We will focus on methods based on parameterization, meaning that we will not consider splitting methods such as the leap-frog or blocking methods. We describe in detail parameterized versions of the following pseudorandom number generators: (i) linear congruential generators, (ii) shift-register generators, and (iii) lagged-Fibonacci generators. We briefly describe the methods, detail some advantages and disadvantages of each method, and recount results from number theory that impact our understanding of their quality in parallel applications. SPRNG was designed around the uniform implementation of different families of parameterized random number generators. We then present a short description of SPRNG. The description contained within this document is meant only to outline the rationale behind and the capabilities of SPRNG. Much more information, including examples and detailed documentation aimed at helping users with putting and using SPRNG on scalable systems is available at the URL: `http://sprng.cs.fsu.edu/RNG`. In this description of SPRNG we discuss the random number generator library as well as the suite of tests of randomness that is an integral part of SPRNG. Random number tools for parallel Monte Carlo applications must be subjected to classical as well as new types of empirical tests of randomness to eliminate generators that show defects when used in scalable environments.

Categories and Subject Descriptors: G.4 [**Mathematics of Computing**]: Mathematical Software—*Algorithm design and analysis, Documentation, Efficiency, Parallel and vector implementations, Reliability and robustness*; D.3.2 [**Software**]: Language Classifications—*C, C++, Fortran*

General Terms: Algorithms, Design, Documentation, Experimentation, Performance, Reliability, Standardization

Additional Key Words and Phrases: Parallel random number generators, random number software, Linear congruential generator, Lagged Fibonacci generator, Random number tests

The SPRNG software was developed with funding from DARPA Contract Number DABT63-95-C-0123 for ITO: Scalable Systems and Software, entitled *A Scalable Pseudorandom Number Generation Library for Parallel Monte Carlo Computations.* This project was a collaboration between David Ceperley, Lubos Mitas, Faisal Saied and Ashok Srinivasan of the University of Illinois at Urbana-Champaign and the first author's research group at the University of Southern Mississippi and the Florida State University.
SPRNG is available from http://sprng.cs.fsu.edu/RNG.

1. INTRODUCTION

Monte Carlo computations have in the past and continue to consume a large fraction of all the available high-performance computing cycles. While much work has been done on numerical software in areas such as numerical linear algebra and the numerical solution of differential equations, there is a dearth of good software tools for Monte Carlo computations, despite their ubiquity and importance. In particular, defects in pseudorandom number generators can cause erroneous results in the Monte Carlo computation. To make matters worse, many of the popular pseudorandom number generators in use, particularly those supplied by vendors, have been found to be defective in the context of real applications. Parallelism further complicates this state of affairs. The random number software we describe in detail below was developed to rectify this situation. Apart from the implementation of good generators, we have designed a user-friendly, standard interface which others can use for their implementation of new **p**arallel **p**seudo**r**andom **n**umber **g**enerators (PPRNGs).

Monte Carlo applications are widely perceived as embarrassingly parallel.[1] The truth of this notion depends, to a large extent, on the quality of the parallel random number generators used. It is widely assumed that with N processors executing N copies of a Monte Carlo calculation, the pooled result will achieve a variance N times smaller than a single instance of this calculation in the same amount of time. This is true only if the results in each processor are statistically independent. In turn, this will be true only if the streams of random numbers generated in each processor are independent.

Here we present an overview of the **S**calable **P**arallel **R**andom **N**umber **G**enerators (`SPRNG`) library. This library was designed to support parallel Monte Carlo applications on scalable and distributed architectures. We begin by briefly presenting several methods of parallel pseudorandom number generation and discuss pros and cons for each method. If the reader is interested in background material on plain old serial pseudorandom number generation consult the following references by Knuth [Knuth 1998], L'Ecuyer [L'Ecuyer 1990; L'Ecuyer 1994; L'Ecuyer 1998], Niederreiter [Niederreiter 1992], and Park and Miller [Park and Miller 1988], and Tezuka [Tezuka 1995], while a good overview of parallel pseudorandom number generation can be found in a recent work by the present article's author(s) [Mascagni 1999b; Mascagni 1999a; Srinivasan et al. 1999a]. Finally, general references on testing PPRNGs can be found in [Cuccaro et al. 1995; De Matteis and Pagnutti 1995; Entacher 1998; Vattulainen 1999].

In our parallel pseudorandom number generation review we are interested, exclusively, in methods for obtaining PPRNGs via parameterization. The exact meaning of parameterization depends on the type of PRNG under discussion, but we wish to distinguish parameterization from splitting methods. We will not be considering the production of parallel streams of pseudorandom numbers by taking

[1]Monte Carlo enthusiasts prefer the term "naturally parallel" to the somewhat derogatory "embarrassingly parallel" coined by computer scientists.

substreams from a single, long-period PRNG. For readers interested in splitting methods and the consequences of using split streams in parallel please consult the works by Deák [Deák 1990], De Matteis and Pagnutti [De Matteis and Pagnutti 1990c; De Matteis and Pagnutti 1990a; De Matteis and Pagnutti 1990b], Frederickson *et al.* [Frederickson et al. 1984], and L'Ecuyer and Côté [L'Ecuyer and Côté 1991]. In general, we seek to determine a parameter in the underlying recursion of the PRNG that can be varied. Each valid value of this parameter will lead to a recursion that produces a unique, full-period stream of pseudorandom numbers. We then discuss efficient means to specify valid parameter values and consider these choices in terms of the quality of the pseudorandom numbers produced. Quality here refers not only to the randomness properties of the individual streams of random numbers but on the correlation properties between streams that are used in parallel.

We then describe the `SPRNG` library in detail. We present the rationale for the design of `SPRNG`, outline the overall design methodology, which is based on full-period parameterized random number generators, and then detail the suite of randomness tests that is included in `SPRNG`. In our selection of random number generators to parameterize and include in `SPRNG`, we utilized extensive tests of randomness to empirically validate and refine our choices. Because our intention was to make `SPRNG` extensible to new types of PRNGs, we have included this suite of tests in our release of `SPRNG`. This suite includes standard single-processor tests of randomness, parallel versions of these same tests, and several inherently parallel tests. Among the inherently parallel tests are several physically-based tests that stress generators with known computations of physical quantities that have been shown to be particularly sensitive to defects in random number generators.

The plan of the paper is as follows. In §2 we present an extensive overview of parallel pseudorandom number generation mostly viewed from the parameterization point of view. In §2.1, we present two methods for parameterizing **l**inear **c**ongruential **g**enerator**s** (LCGs). In §2.2 we present a parameterization of another linear method: **s**hift-**r**egister **g**enerator**s** (SRGs). This parameterization is analogous to one of the LCG parameterizations presented in §2.1. The SRG has been implemented but not released into `SPRNG` to date. The reason being that performance of the SRGs is suboptimal, and we wish to tune the code for performance before releasing it into `SPRNG`. However, we plan to incorporate both this version of the SRG as well as the famous "Mersenne Twister" SRG of Matsumoto and Kurita, [Matsumoto and Kurita 1992; Matsumoto and Nishimura 1998], into `SPRNG` in the very near future. In §2.3 we consider the parallel parameterization of so-called lagged-Fibonacci generators. In §3, we describe the `SPRNG` library, a comprehensive tool for parallel and distributed pseudorandom number generation developed by the authors. In this description we also give some examples on how to use `SPRNG` in serial and parallel codes. We then, in §4, describe the suite of tests of randomness that is an integral part of `SPRNG`. Finally, in §5 we conclude and comment on some new directions for `SPRNG`.

2. PARALLEL PSEUDORANDOM NUMBER GENERATION

In this next, rather extensive, section we will look at several methods for parallel pseudorandom number generation. Most of the methods we will present will be based on some kind of parameterization of the generators.

2.1 Linear Congruential Generators

The most commonly used generator for pseudorandom numbers is the LCG. The LCG was first proposed for use by Lehmer [Lehmer 1949], and is referred to as the Lehmer generator in the early literature. The linear recursion underlying LCGs is:

$$x_n = ax_{n-1} + b \pmod{m}. \tag{1}$$

When the multiplier, a, additive constant, b, and modulus, m, are chosen appropriately one obtains a purely periodic sequence with period as long as $Per(x_n) = 2^k$, when m is a power-of-two, and $Per(x_n) = m-1$, when m is prime. It is well known that s-tuples made up from LCGs lie on lattices composed of a family of parallel hyperplanes, Marsaglia [Marsaglia 1968; Marsaglia 1972]. The x_n's in Eq. (1) are integer residues modulo m, and a uniform pseudorandom number in [0,1] is produced via $z_n = x_n/m$, and the initial value of the LCG, x_0, is often called the seed.

The most important parameter of an LCG is the modulus, m. Its size constrains the period, and for implementational reasons it is always chosen to be either prime or a power-of-two. Based on which type of modulus is chosen, there is a different parameterization method. When m is prime, a method based on using the multiplier, a, as the parameter has been proposed. The rationale for this choice is outlined in Mascagni [Mascagni 1998], and leads to several interesting computational problems.

2.1.1 Prime Modulus

Given we wish to parameterize a when m is prime, we must determine first the family of permissible a's. A condition on a when m is prime to obtain the maximal period (of length $m-1$ in this case) is that a must be a primitive element modulo m, Knuth [Knuth 1998].[2] Given primitivity, one can use the following fact: if a and α are primitive elements modulo m then $\alpha = a^i \pmod{m}$ for some i relatively prime to $\phi(m)$. Note that when m is prime that $\phi(m) = m-1$. Thus a single, reference, primitive element, a, and an explicit enumeration of the integers relatively prime to $m-1$ furnish an explicit parameterization for the jth primitive element, a_j as $a_j = a^{\ell_j} \pmod{m}$ where ℓ_j is the jth integer relatively prime to $m-1$. Given an explicit factorization of $m-1$, Brillhart *et al.* [Brillhart et al. 1988], efficient algorithms for computing ℓ_j can be found in a recent work of the author [Mascagni 1998]. An interesting open question in this regard is whether the overall efficiency of this PPRNG is minimized by choosing the prime modulus to minimize the cost of computing ℓ_j or to minimize the cost of modular multiplication modulo m.

Given this scheme there are some positive and negative features to be mentioned. A motivation for this scheme is that a common theoretical measure of the correlation among parallel streams predicts little correlation. This measure is based on exponential sums. Exponential sums are of interest in many areas of number theory. We define the exponential sum for the sequence of residues modulo m, $\{x_n\}_{n=0}^{k-1}$,

[2]An integer, a, is primitive modulo m if the set of integers $\{a^i \pmod{m} | 1 \le i \le m-1\}$ equals the set $\{1 \le i \le m-1\}$.

as:

$$C(k) = \sum_{n=0}^{k-1} e^{\frac{2\pi i}{m} x_n}, \quad (i = \sqrt{-1}). \tag{2}$$

If the x_n are periodic and k is the period, then Eq. (2) is called a full-period exponential sum. If x_n is periodic and k is less than the full period, then Eq. (2) is a partial-period exponential sum. Examining Eq. (2) shows it to be a sum of k quantities on the unit circle. A trivial upper bound is thus $|C(k)| \leq k$. If the sequence $\{x_n\}$ is indeed uniformly distributed, then we would expect $|C(k)| = O(\sqrt{k})$, Kuipers and Niederreiter [Kuipers and Niederreiter 1974]. Thus the desire is to show that exponential sums of interest are neither too big nor too small to reassure us that the sequence in question is theoretically equidistributed.

Since we are interested in studying sequences for use in parallel, we must consider the cross-correlations among the sequences to be used on different processors. If $\{x_n\}$ and $\{y_n\}$ are two sequences of interest then their exponential sum cross-correlation is given by:

$$C(i,j,k) = \sum_{n=0}^{k-1} e^{\frac{2\pi i}{m}(x_{i+n} - y_{j+n})}, \quad (i = \sqrt{-1}). \tag{3}$$

Here the sum has k terms and begins with x_i and y_j.

In a previous work we only considered full-period exponential sum cross-correlation for studying these issues for a different recursion, Pryor *et al.* [Pryor et al. 1994]. We will take the same approach here. Suppose we have j full-period LCGs defined by $x_{k_n} = a^{\ell_i} x_{k_{n-1}} \pmod{m}$, $0 \leq k < j$. All of the pairwise full-period exponential sum cross-correlations are known to satisfy, Schmidt [Schmidt 1976]:

$$|C(m)| \leq \left(\left[\max_k \ell_k \right] - 1 \right) \sqrt{m}. \tag{4}$$

The choice of the exponents, ℓ_k, that minimizes Eq. (4) is to make ℓ_j the jth integer relatively prime to $m-1$. This necessitates an algorithm to compute this jth integer relatively prime to an integer with known factorization, $m-1$. This is discussed at great length in Mascagni [Mascagni 1998]; however, two important open questions remain: (1) is it more efficient overall to choose m to be amenable to fast modular multiplication or fast calculation of the jth integer relatively prime to $m-1$, and (2) does the good interstream correlation of Eq. (4) also ensure good intrastream independence via the spectral test? The first of these questions is of practical interest to performance, the second; however, if answered negatively, makes such techniques less attractive for parallel pseudorandom number generation.

2.1.2 Power-of-two Modulus

An alternative way to use LCGs to make a PPRNG is to parameterize the additive constant in Eq. (1) when the modulus is a power-of-two, i.e., to $m = 2^k$ for some integer $k > 1$. This is a technique first proposed by Percus and Kalos [Percus and Kalos 1989], to provide a PPRNG for the NYU Ultracomputer. It has some interesting advantages over parameterizing the multiplier; however, there are some considerable disadvantages in using power-of-two modulus LCGs.

The parameterization chooses a set of additive constants $\{b_j\}$ that are pairwise relatively prime, i.e. $\gcd(b_i, b_j) = 1$ when $i \neq j$. A prudent choice is to let b_j be the jth prime. This both ensures the pairwise relative primality and is the largest set of such residues. With this choice certain favorable interstream properties can be theoretically derived from the spectral test [Percus and Kalos 1989]. However, this choice necessitates a method for the difficult problem of computing the jth prime. In their paper, Percus and Kalos do not discuss this aspect of their generator in detail, partly due to the fact that they expect to provide only a small number of PRNGs. When a large number of PPRNGs are to be provided with this method, one can use fast algorithms for the computation of $\pi(x)$, the number of primes less than x, Deleglise and Rivat [Deleglise and Rivat 1996], Lagarias, Miller, and Odlyzko [Lagarias et al. 1985] . This is the inverse of the function which is desired, so we designate $\pi^{-1}(j)$ as the jth prime. The details of such an implementation need to be specified, but a very related computation for computing the jth integer relatively prime to a given set of integers is given in Mascagni [Mascagni 1998]. It is believed that the issues for computing $\pi^{-1}(j)$ are similar. At present, this is implemented in `SPRNG` with a table of primes, and to date `SPRNG` users have found this to be fast enough and to provide a sufficient number of parallel streams.

One important advantage of this parameterization is that there is an interstream correlation measure based on the spectral test that suggests that there will be good interstream independence. Given that the spectral test for LCGs essentially measures the quality of the multiplier, this sort of result is to be expected. A disadvantage of this parameterization is that to provide a large number of streams, computing $\pi^{-1}(j)$ will be necessary. Regardless of the efficiency of implementation, this is known to be a difficult computation with regards to its computational complexity. Finally, one of the biggest disadvantages to using a power-of-two modulus is the fact the least significant bits of the integers produced by these LCGs have extremely short periods. If $\{x_n\}$ are the residues of the LCG modulo 2^k, with properly chosen parameters, $\{x_n\}$ will have period 2^k. However, $\{x_n \pmod{2^j}\}$ will have period 2^j for all integers $0 < j < k$, Knuth [Knuth 1998]. In particular, this means the least-significant bit of the LCG with alternate between 0 and 1. This is such a major short coming, that it motivated us to consider parameterizations of prime modulus LCGs as discussed in §2.1.1.

2.2 Shift-Register Generators

Shift **r**egister **g**enerators (SRGs) are linear recursions modulo 2, Golomb [Golomb 1982], Lewis and Payne [Lewis and Payne 1973], and Tausworthe [Tausworthe 1965], of the form:

$$x_{n+k} = \sum_{i=0}^{k-1} a_i x_{n+i} \pmod{2}, \tag{5}$$

where the a_i's are either 0 or 1. An alternative way to describe this recursion is to specify the kth degree binary characteristic polynomial, Lidl and Niederreiter [Lidl and Niederreiter 1986]:

$$f(x) = x^k + \sum_{i=0}^{k-1} a_i x^i \pmod{2}. \tag{6}$$

To obtain the maximal period of $2^k - 1$, a sufficient condition is that $f(x)$ be a primitive kth degree polynomial modulo 2. If only a few of the a_i's are 1, then Eq. (5) is very cheap to evaluate. Thus people often use known primitive trinomials to specify SRG recursions. This leads to very efficient, two-term, recursions.

There are two ways to make pseudorandom integers out of the bits produced by Eq. (5). The first, called the digital multi-step method, takes successive bits from Eq.(5) to form an integer of desired length. Thus, with the digital multi-step method, it requires n iterations of Eq. (5) to produce a new n-bit pseudorandom integer. The second method, called the generalized feedback shift-register, creates a new n-bit pseudorandom integer for every iteration of Eq. (5). This is done by constructing the n-bit word from x_{n+k} and $n-1$ other bits from the k bits of SRG state. While these two methods seem different, they are very related, and theoretical results for one always hold for the other. One way to parameterize SRGs is analogous to the LCG parameterization discussed in §2.1.1. There we took the object that made the LCG full-period, the primitive root multiplier, and found a representation for all of them. Using this analogy we identify the primitive polynomial in the SRG as the object to parameterize. We begin with a known primitive polynomial of degree k, $p(x)$. It is known that only certain decimations of the output of a maximal-period shift register are themselves maximal and unique with respect to cyclic reordering, Lidl and Niederreiter [Lidl and Niederreiter 1986]. We seek to identify those. The number of decimations that are both maximal-period and unique when $p(x)$ is primitive modulo 2 and k is a Mersenne exponent is $\frac{2^k-2}{k}$. If a is a primitive root modulo the prime 2^k-1, then the residues $a^i \pmod{2^k-1}$ for $i=1$ to $\frac{2^k-2}{k}$ form a set of all the unique, maximal-period decimations. Thus we have a parameterization of the maximal-period sequences of length 2^k-1 arising from primitive degree k binary polynomials through decimations.

The entire parameterization goes as follows. Assume the kth stream is required, compute $d_k \equiv a^k \pmod{2^k-1}$ and take the d_kth decimation of the reference sequence produced by the reference primitive polynomial, $p(x)$. This can be done quickly with polynomial algebra. Given a decimation of length $2k+1$, this can be used as input the Berlekamp-Massey algorithm to recover the primitive polynomial corresponding to this decimation. The Berlekamp-Massey algorithm finds the minimal polynomial that generates a given sequence, Massey [Massey 1969] in time linear in k.

This parameterization is relatively efficient when the binary polynomial algebra is implemented correctly. However, there is one major drawback to using such a parameterization. While the reference primitive polynomial, $p(x)$, may be sparse, the new polynomials need not be. By a sparse polynomial we mean that most of the a_i's in Eq. (5) are zero. The cost of stepping Eq. (5) once is proportional to the number of non-zero a_i's in Eq. (5). Thus we can significantly increase the bit-operational complexity of a SRG in this manner.

The fact that the parameterization methods for prime modulus LCGs and SRGs are so similar is no accident. Both are based on maximal period linear recursions over a finite field. Thus the discrepancy and exponential sum results for both the types of generators are similar, Niederreiter [Niederreiter 1992]. However, a result for SRGs analogous to that in Eq. (4) is not known. It is open whether or not such a cross-correlation result holds for SRGs, but it is widely thought to.

2.3 Lagged-Fibonacci Generators

In the previous sections we have discussed generators that can be parallelized by varying a parameter in the underlying recursion. In this section we discuss the **a**dditive **l**agged-**F**ibonacci **g**enerator (ALFG): a generator that can be parameterized through its initial values. The ALFG can be written as:

$$x_n = x_{n-j} + x_{n-k} \pmod{2^m}, \quad j < k. \tag{7}$$

In recent years the ALFG has become a popular generator for serial as well as scalable parallel machines, Makino [Makino 1994]. In fact, the generator with $j = 5$, $k = 17$, and $m = 32$ was the standard PPRNG in Thinking Machines Connection Machine Scientific Subroutine Library. This generator has become popular for a variety of reasons: (1) it is easy to implement, (2) it is cheap to compute using Eq. (7), and (3) the ALFG does well on standard statistical tests, Marsaglia [Marsaglia 1985].

An important property of the ALFG is that the maximal period is $(2^k - 1)2^{m-1}$. This occurs for very specific circumstances, Brent [Brent 1994] and Marsaglia and Tsay [Marsaglia and Tsay 1985], from which one can infer that this generator has $2^{(k-1)\times(m-1)}$ different full-period cycles, Mascagni *et al.* [Mascagni et al. 1995]. This means that the state space of the ALFG is toroidal, with Eq. (7) providing the algorithm for movement in one of the torus dimension. It is clear that finding the algorithm for movement in the other dimension is the basis of a very interesting parameterization. Since Eq. (7) tells us how to cycle over the full period of the ALFG, one must find a seed that is not in a given full-period cycle to move in the second dimension. The key to moving in this second dimension is to find an algorithm for computing seeds in any given full-period cycle.

A very elegant algorithm for movement in this second dimension is based on a simple enumeration as follows. One can prove that the initial seed, $\{x_0, x_1, \ldots, x_{k-1}\}$, can be bit-wise initialized using the following template:

$$\begin{array}{cccc|c|l} \text{m.s.b.} & & & & \text{l.s.b.} & \\ b_{m-1} & b_{m-2} & \ldots & b_1 & b_0 & \\ \hline \blacksquare & \blacksquare & \ldots & 0 & 0 & x_{k-1} \\ 0 & \blacksquare & \ldots & \blacksquare & 0 & x_{k-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \\ \blacksquare & 0 & \ldots & \blacksquare & 0 & x_1 \\ \blacksquare & \blacksquare & \ldots & \blacksquare & 1 & x_0 \end{array} \tag{8}$$

Here each square is a bit location to be assigned. Each unique assignment gives a seed in a provably distinct full-period cycle, Mascagni *et al.* [Mascagni et al. 1995]. Note that here the least-significant bits, b_0 are specified to be a fixed, non-zero, pattern. If one allows an $O(k^2)$ precomputation to find a particular least-significant-bit pattern then the template is particularly simple:

$$\begin{array}{cccc|c|c} \text{m.s.b.} & & & & \text{l.s.b.} & \\ b_{m-1} & b_{m-2} & \dots & b_1 & b_0 & \\ \hline \blacksquare & \blacksquare & \dots & \blacksquare & b_{0\,k-1} & x_{k-1} \\ \blacksquare & \blacksquare & \dots & \blacksquare & b_{0\,k-2} & x_{k-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \\ \blacksquare & \blacksquare & \dots & \blacksquare & b_{0\,1} & x_1 \\ \hline 0 & 0 & \dots & 0 & b_{0\,0} & x_0 \end{array} \tag{9}$$

Given the elegance of this explicit parameterization, one may ask about the exponential sum correlations between these parameterized sequences. It is known that certain sequences are more correlated than others as a function of the similarity in the least-significant bits in the template for parameterization, Mascagni *et al.* [Mascagni et al. 1995]. However, it is easy to avoid all but the most uncorrelated pairs in a computation, Pryor *et al.* [Pryor et al. 1994]. In this case there is extensive empirical evidence that the full-period exponential sum correlation between streams is $O(\sqrt{(2^k-1)2^{m-1}})$, the square root of the full-period. This is essentially optimal. Unfortunately, there is no analytic proof of this result, and improvement of the best known analytic result, Mascagni *et al.* [Mascagni et al. 1995], is an important open problem in the theory of ALFGs.

Another advantage of the ALFG is that one can implement these generators directly with floating-point numbers to avoid the constant conversion from integer to floating-point that accompanies the use of other generators. This is a distinct speed improvement when only floating-point numbers are required in the Monte Carlo computation. However, care must be taken to maintain the identity of the corresponding integer recursion when using the floating-point ALFG in parallel to maintain the uniqueness of the parallel streams. A discussion of how to ensure fidelity with the integer streams can be found in Brent [Brent 1992].

An interesting cousin of the ALFG is the **m**ultiplicative **l**agged-**F**ibonacci **g**enerator (MLFG). It is defined by:

$$x_n = x_{n-j} \times x_{n-k} \pmod{2^m}, \quad j < k. \tag{10}$$

While this generator has a maximal-period of $(2^k-1)2^{m-3}$, which is a quarter the length of the corresponding ALFG, Marsaglia and Tsay [Marsaglia and Tsay 1985], it has empirical properties considered to be superior to ALFGs, Marsaglia [Marsaglia 1985]. Of interest for parallel computing is that a parameterization analogous to that of the ALFG exists for the MLFG, Mascagni [Mascagni 1997] and Coddington [Coddington 1996; Coddington 1997].

3. SPRNG

The `SPRNG` library is currently in it's first, full, `Version 1.0` release. Moreover `SPRNG` is now supported and maintained by NCSA under their high-performance software activities funded by the NSF under PACI. In addition, there has been considerable interest from most of the high-performance computing vendors in using `SPRNG` as a common, parallel pseudorandom number generation library on their machines. Thus `SPRNG`, itself, will be a lasting contribution to mathematical software for parallel Monte Carlo computations.

SPRNG is designed to use parameterized pseudorandom number generators to provide random number streams to parallel processes. SPRNG includes the following:

—Several, qualitatively distinct, well tested, scalable PRNGs
—Initialization without interprocessor communication
—Reproducibility by using the parameters to index the streams
—Reproducibility controlled by a single "global" seed
—Minimization of interprocessor correlation with the included generators
—A uniform C, C++, FORTRAN, and MPI interface
—Extensibility
—An integrated test suite including physical tests

The decision to use parameterized generators was based on work of the author in parameterizing several different, common, PRNGs to provide full-period streams of random numbers for each, unique, parameter value. These generators then formed the core of the generators currently available in SPRNG:

—Additive lagged-Fibonacci: $x_n = x_{n-r} + x_{n-s} \pmod{2^m}$
—Multiplicative lagged-Fibonacci: $x_n = x_{n-r} \times x_{n-s} \pmod{2^m}$
—Prime modulus multiplicative congruential: $x_n = a x_{n-1} \pmod{m}$
—Power-of-two modulus linear congruential: $x_n = a x_{n-1} + b \pmod{2^m}$
—Combined multiple recursive generator: $z_n = x_n + y_n \times 2^{32}$, where x_n is a linear congruential generator modulo 2^{64} and y_n satisfies $y_n = 107374182 y_{n-1} + 104480 y_{n-5} \pmod{2147483647}$

All the above generators can be thought of as being parameterized by a simple integer valued function $f(\cdot)$, where $f(i)$ gives the appropriate parameter for the ith random number stream. Given this uniformity, the random number streams are mapped onto the binary tree through the canonical enumeration via the index i. This allows us to take the parameterization and use it to produce new streams from existing streams without the need for interprocessor communication. We accomplish this by allowing a given stream access only to those streams associated with the subtree rooted at the given stream. This can be used to automatically manage static and dynamic creation of streams, and prohibits reuse of streams. To permit a calculation to be redone with different random numbers, we can apply a mixing function $p_s(\cdot)$ so that we map the streams onto the binary tree via the index $p_s(i)$ instead of just i. The function $p_s(\cdot)$ is a permutation parameterized by the global seed s. Different values of s give different permutations and thus map the streams onto the binary tree in different yet distinct ways. In our initial work with parallelizing ALFGs, we built $p_s(\cdot)$ up from an SRG, where s was a 31-bit seed to the same sized SRG. We found that the SRG gave unexpected interstream correlations and changed over to an analogous LCG, which eliminated the correlations. Because of this experience we feel that a very interesting area for future research is in characterizing and implementing good permutation functions.

SPRNG was also designed to be flexible, and to be as easy to use as possible. The Monte Carlo community is very conservative, and many groups use PRNGs that have been handed down the generations (sometimes all the way back to Lehmer or Metropolis!). Thus we not only developed the library in collaboration with a member of this conservative community, but we added the ability to extend the library with a user supplied generator. Thus a user may add their own PRNG by rewriting two dummy SPRNG functions and recompiling SPRNG. This then gives a user access to their own generator within the SPRNG parallel infrastructure. This is a powerful capability, and our own implementational experience has shown that any implementation must be thoroughly tested, empirically, to prevent unforeseen correlations within streams (we found such unanticipated correlations ourselves in very carefully thought out implementations). Thus SPRNG includes a comprehensive testing suite to validate new generators. Together, the extensibility and testing suite aids both users wanting to implement their own generators in parallel, and provides library developers a powerful rapid prototyping tool.

Through the default generators, SPRNG is a tool for parallel pseudorandom number generation. The results obtained are also reproducible, and SPRNG provides a simple way to run on distributed-memory parallel machines using popular languages and parallel paradigms and supports distribution on a heterogeneous collection of machines.[3] When a different PRNG is desired, e.g. when a particular PRNG is thought to give spurious results in a given application, a qualitatively different generator can replace the original by merely recompiling the user program with calls to the appropriate new SPRNG generator.[4] Finally, new PRNGs can be incorporated into SPRNG with little more than coding the generation and initialization routines and recompiling SPRNG.

It is important to remember that SPRNG was designed initially to provide support for distributed memory multiprocessor systems. As such, certain parallel random number tasks are very easy with SPRNG while others require some work. As an example of the latter, suppose one wants to use SPRNG to produce a code that uses the same random numbers on any number of processors. To do so, one must begin with a Monte Carlo computation that has an *a priori* decomposition of the random numbers used into explicit substreams. This is because each SPRNG generator is based on a parameterized generator, and so within a certain type of generator it is usually not the case that random number substreams are part of a single generator's full period. Thus, the easiest way to produce a code that uses the same random numbers on any number of processors is to break up the random number use into discrete, known blocks and use the blocks in the order that they are provided by SPRNG, independently of the number of processors. In fact, this is a very easy job for most Monte Carlo applications, and once such a code is written it may be efficiently and reproducibly executed on a wide variety of parallel, clustered, and distributed machines.

[3]In fact, the developers of CONDOR, a distributed computing tool, plan to incorporate SPRNG directly into CONDOR to make CONDOR a comprehensive tool for Monte Carlo on distributed heterogeneous collections of machines, see Litzkow *et al.* [Litzkow et al. 1998]. Preliminary work by one of the authors and a colleague has already been accomplished, [Zhou and Mascagni 1999].

[4]Version 1.0 of SPRNG has separate generators in one of several separate SPRNG libraries. The currently available version of SPRNG, version 2.0, has all generators in a single library.

Using SPRNG: Some Examples

The SPRNG library has extensive on-line documentation available on the SPRNG web pages. Thus the curious reader is referred to these web pages for a detailed tutorial and examples. In this article we present the bare minimum required for a user to use SPRNG via the "simple" interface. Below we show the definition of the init_sprng routine along with a description of all of the inputs required to call init_sprng. This is the routine that one calls to initialize the various random number streams for parallel use.

```
int *init_sprng(int streamnum, int nstreams, int seed, int param)
SPRNG_POINTER init_sprng(integer streamnum, integer nstreams,
                          integer seed, integer param)
```

—init_sprng initializes random number *streams*.

—streamnum is the stream number and is typically the process number and must be in [0,nstreams-1].

—nstreams is the number of different streams that will be initialized across all the processes.

—seed is the *seed* to the generators. The seed is not the starting state of the sequence; rather, it is an encoding of the starting state. It is acceptable (and recommended) to use the same seed for all the streams.

—param selects the appropriate parameters (for example, the multiplier for a Linear Congruential Generator or the lag for a Lagged Fibonacci Generator).

—init_sprng returns the *ID* of the stream.

Below we have a simple "C" program that shows how to initialize several SPRNG PRNGs in parallel using MPI. This example shows how east it is to replace existing PRNGs with those from SPRNG. Note, in the subsequent example, we use SPRNG's "Simple" interface which only has two arguments in its init_sprng call. In fact, with the "Simple" interface one can avoid calling init_sprng and use the default settings. In our general discussion of init_sprng given above, we show its use using the "Default" interface, which permits the user to specify all four arguments as described.

```
/***************************************************************************/
/*            Demonstrates sprng use with one stream per process           */
/* A distinct stream is created on each process, then prints a few         */
/* random numbers.                                                         */
/***************************************************************************/


#include <stdio.h>
#include <mpi.h>               /* MPI header file                           */

#define SIMPLE_SPRNG           /* simple interface                          */
#define USE_MPI                /* use MPI to find number of processes       */
#include "sprng.h"             /* SPRNG header file                         */
```

```
#define SEED 985456376


main(int argc, char *argv[])
{
  double rn;
  int i, myid;


  /*************************** MPI calls ***********************************/

  MPI_Init(&argc, &argv);                 /* Initialize MPI               */
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);   /* find process id              */

  /************************** Initialization *******************************/

  init_sprng(SEED,SPRNG_DEFAULT);         /* initialize stream            */
  printf("Process %d, print information about stream:\n", myid);
  print_sprng();

  /************************ print random numbers ***************************/

  for (i=0;i<3;i++)
  {
    rn = sprng();       /* generate double precision random number        */
    printf("Process %d, random number %d: %.14f\n", myid, i+1, rn);
  }

  MPI_Finalize();       /* Terminate MPI                                  */
}
```

4. SPRNG TEST SUITE

The results of **M**onte **C**arlo (MC) computations can be adversely affected by defects (essentially, correlations) in the random number sequences used. A PPRNG must be tested for two types of correlations – (i) intrastream correlation (correlation between numbers in the same stream) , as for any serial generator, and (ii) interstream correlation for correlations between random number streams on different processes. Since bounds on these correlations are difficult to prove mathematically, large empirical tests are necessary. Many of the popular PRNGs in use today were tested when computational power was much lower, and hence they were evaluated with much smaller test sizes. Defects were later discovered in several such generators when used in simulations [Ferrenberg et al. 1992; Grassberger 1993]. The SPRNG generators, on the other hand, have passed some of the largest empirical random number tests ever undertaken; many of the statistical tests being a few orders of magnitude larger than previous records.

SPRNG therefore comes with a test suite to verify the quality of serial and parallel random number sequences. We corrected all the defects in our earlier generators so that they pass these standard PRNG tests, making the SPRNG generators some of the best tested available. The SPRNG test suite can also be used by others to test their own generators. We consider this to be important, because defects are often observed in generators as the size of the simulations increase. As computers become faster, new generators will inevitably have to be developed and tested.

The SPRNG test suite consists of (i) statistical tests and (ii) physically-based tests. The statistical tests are designed so that the expected value of some test statistic is known for an independent identically distributed random sample from the uniform distribution. The empirically generated random number stream is then subjected to the same test, and the statistic obtained is compared against the expected value. It is also necessary to verify the quality of a PRNG by using it in real applications. Thus we also include physically-based tests which use random numbers in a manner similar to that in a real application, except that the exact solution is known. The advantage of the statistical tests are that they are usually much faster than the physically-based randomness tests. On the other hand, the latter use random numbers in the same manner as real applications, and can thus be considered more representative, and they test correlations of more numbers at a time.

4.1 Statistical Tests

The tests described by Knuth [Knuth 1998] and those implemented in the DIEHARD package [Marsaglia] are considered de facto standards for serial PRNGs. The SPRNG test suite includes all the tests described by Knuth. We have modified these tests to test for parallel correlation in the following manner: we interleave several streams to form a new stream, and then test this new stream with conventional tests as shown in figure 1.

We form several such new streams, and test several blocks of random numbers from each stream. Usually the result of the test for each block is a Chi-square value. We take the Chi-square statistics for all the blocks and use the **K**olmogorov-**S**mirnov (KS) test to verify that they are distributed according to the Chi-square distribution. If the KS percentile is between 2.5% and 97.5%, then the test is passed by the random number generator.

The tests in the SPRNG test suite take several arguments. The first few arguments are common to all the statistical tests, and are explained below. The test specific arguments will be explained later. We also have a Java Test Wizard that helps users specify the test arguments in the correct order. The SPRNG tests are called as follows:

```
test.lib nstreams ncombine seed param nblocks skip test_args
```

where the name of the executable *test.lib* is formed by concatenating the name of the test and the random number library from which the random numbers are generated. For example:

```
equidist.lcg 4 2 0 0 3 1 2 100
mpirun -np 2 equidist.lcg 4 2 0 0 3 1 2 100
```

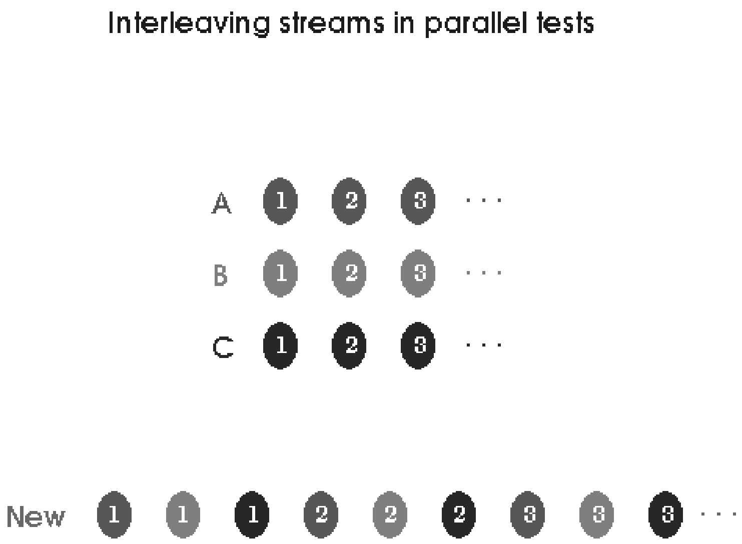

Fig. 1. We interleave streams to form a new stream, and test this stream with conventional serial tests.

perform the equidistribution test with the 48 bit Linear Congruential Generator with prime addend in a serial and parallel machine respectively.

The argument *ncombine* (= 2 in our example) indicates the number of streams we interleave to form a new stream. We form *nstreams* (= 4) such new streams and test *nblocks* (= 3) blocks of random numbers from each new stream. The argument *seed* (= 0) is the encoded seed to the random number generator, and *param* (= 0) is the parameter to the generator. The argument *skip* (= 1) indicates how many random numbers we skip after testing a block before we start a test on the next block. The rest of the arguments in our example are specific to that test. Note that we can perform tests on individual streams (tests for intra-stream correlations) by setting *ncombine* to 1. The meaning of the test parameters is clarified in figure 2

The result of the example given above are as follows:

```
sprng/TESTS:sif% mpirun -np 2 equidist.lcg 4 2 0 0 3 1 2 100
equidist.lcg 4 2 0 0 3 1 2 100
Result: KS value = 0.601752
KS value prob = 17.50
```

The KS *value prob* line gives the KS percentile for the entire set of tests. Since it is between 2.5% and 97.5%, we consider this example to have passed. It should be noted that the real tests are much larger than this simple example.

Note: When we state that a particular test is parallel, we are referring to the fact that it can be used to test for correlations between streams. We do not mean that it actually runs on multiple processors. All the SPRNG statistical tests can run either on a single processor or on multiple processors.

We next briefly describe each test followed by its test specific arguments. We also

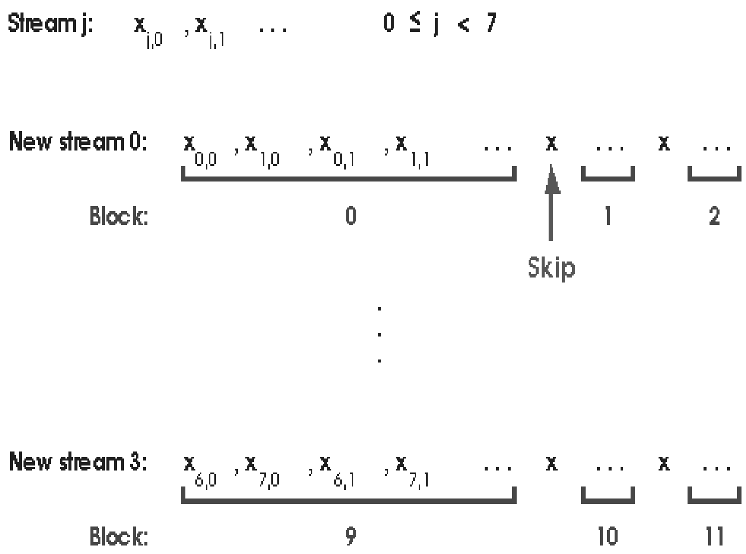

Fig. 2. Illustration of the test parameters from the example given above.

give the number of random numbers tested and asymptotic memory requirements (in bytes, assuming an integer is 4 bytes and double precision is 8 bytes). This should helps users estimate the time required for their calculations from smaller sample runs.

The details concerning these tests are presented in Knuth [Knuth 1998], unless we mention otherwise.

(1) *Collisions test:* $n \ \log md \ \log d$

We concatenate the *logd* most significant bits from *logmd* random integers to form a new $\log_2 m = \log md * \log d$ bit random integer. We form $n \ll m$ such numbers. A collision is said to have occurred each time some such number repeats. We count the number of collisions and compare with the expected number. This test thus checks for absence of *logd*-dimensional correlation. It is one of the most effective tests among those proposed by Knuth.

Number of Random numbers tested: $n * \log md$
Memory: $8 * nstreams * nblocks + 4 * n + 2^{\log md * \log d}$

(2) *Coupon collector's test:* $n \ t \ d$

We generate random integers in $[0, d-1]$. We then scan the sequence until we find at least one instance of each of the d integers, and note the length of the segment over which we found this complete set. For example, if $d = 3$ and the sequence is: 0, 2, 0, 1, ..., then the length of the first segment over which we found a complete set of integers is 4. We continue from the next position in the sequence until we find n such complete sets. The distribution of lengths of the segments is compared against the expected distribution. In our analysis, we lump segments of length $> t$ together.

Number of random numbers tested: $n * d * \log d$
Memory: $8 * nstreams * nblocks + 4 * d + 16 * (t - d + 1)$

(3) *Equidistribution test: d n*

We generate random integers in $[0, d-1]$ and check whether they come from a uniform distribution, that is, if each of the d numbers has equal probability.

Number of random numbers tested: n
Memory: $8 * nstreams * nblocks + 16 * d$

(4) *Gap test: t a b n*

We generate floating point numbers in $(0, 1)$ and note the gap in the sequence between successive appearances of numbers in the interval $[a, b]$ in $(0, 1)$. For example, if $[a, b] = [0.4, 0.7]$ and the sequence is: $0.1,\ 0.5,\ 0.9,\ 0.6, ...$, then the length of the first gap (between the numbers 0.5 and 0.6) is 2. We record n such gaps, and lump gap lengths greater than t together in a category in our analysis.

Number of random numbers tested: $n/(b - a)$
Memory: $8 * nstreams * nblocks + 16 * t$

(5) *Maximum-of-t test: n t*

We generate t floating point numbers in $[0, 1)$ and note the largest number. We repeat this n times. The distribution of this largest number should be as x^t.
Number of random numbers tested: $n * m$
Memory: $8 * nstreams * nblocks + 16 * n$

(6) *Permutations test: m n*

We generate m floating point numbers in $(0, 1)$. We can rank them according to their magnitude; the smallest number is ranked 1, ..., the largest is ranked m. There are $m!$ possible ways in which the ranks can be ordered. For example, if $m = 3$, then the following orders are possible: $(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)$. We repeat this test n times and check if each possible permutations was equally probable.

Number of random numbers tested: $n * m$
Memory: $8 * nstreams * nblocks + 8 * m + 16 * (m!)$

(7) *Poker test: n k d*

We generate k integers in $[0, d-1]$ and count the number of distinct integers obtained. For example if $k = 3, d = 3$ and the sequence is: 0, 1, 1, ..., then the number of distinct integers obtained in the first 3-tuple is 2. We repeat this n times and compare with the expected distribution for random samples from the uniform distribution.

Number of random numbers tested: $n * k$
Memory: $8 * nstreams * nblocks + 0.4 * min(n, k) + 12 * k + 4 * d$

(8) *Runs up test: t n*

We count the length of a "run" in which successive random numbers are non-decreasing. For example if the sequence is: 0.1, 0.2, 0.4, 0.3, then the length of the first run is 3. We repeat this n times and compare with the expected distribution of run lengths for random samples from the uniform distribution. Runs of length greater than t are lumped together during our analysis.

Number of random numbers tested: $1.5 * n$
Memory: $8 * nstreams * nblocks + 16 * t$

(9) *Serial test: d n*

We generate n pairs of integers in $[0, d-1]$. Each of the d^2 pairs should be equally likely to occur.

Number of random numbers tested: $2 * n$
Memory: $8 * nstreams * nblocks + 16 * d * d$

4.1.1 *Inherently parallel tests:.* Unlike the preceding tests which modify sequential tests to test for correlations between streams, the tests mentioned below are inherently parallel. The meaning of the arguments for these tests are slightly different from those for the preceding tests. Since these tests are inherently parallel, we need not interleave streams, and thus the second argument *ncombine* should be set to 1. The first argument *nstreams* is the total number of streams tested. All these streams are tested simultaneously, rather than independently as in the previous case. The rest of the arguments are identical to the previous case.

(1) *Blocking (sum of independent distributions) test: n groupsize*

The central limit theorem states that the sum of *groupsize* independent variables with zero mean and unit variance approaches the normal distribution with mean zero and variance equal to *groupsize*. To test for the independence of random number streams, we form such random variables and check for normality. Clearly, with *groupsize* small, we expect deviation from normality. However, it is well known that 12 uniform random numbers almost completely converge to the normal. We add *groupsize* $>> 100$ random numbers in $(0, 1)$ from each stream to form a sum. We generate n such sums and check if their distribution is normal. (Note: We also computed the exact distribution and determined that the assumption of normality was acceptable for the number of random numbers we added in our tests.)

(2) *Fourier Transform test: n*

We fill a two dimensional array with random numbers. Each row of the array is filled with n random numbers from a different stream. We calculate the Fourier coefficients and compare with the expected values. This test is repeated several times and we check if there are particular coefficients that are repeatedly "bad". The current implementation uses the FFT routine provided by SGI. Users need to modify this test to the FFT routine available on their local machine.

4.2 Physically-based Tests

(1) *Ising model:* For statistical mechanical applications, the two-dimensional Ising model (a simple lattice spin model) is often used since the exact answer for the Energy and Specific Heat are known [Beale 1996]. Since the Ising model is also known to have a phase transition, this system is sensitive to long-range correlations in the PRNG. There are several different algorithms, such as the Metropolis and the Wolff algorithms, that can be used to simulate the Ising model, and the random numbers enter quite differently in each algorithm. Thus this application is very popular in testing random number generators and has often detected defects in generators [Coddington 1994; Coddington 1996; Ferrenberg et al. 1992; Selke et al. 1993; Vattulainen et al. 1994]. We test parallel generators on the Ising model application by assigning a different random number stream for each site.
Our current Ising model codes run on a single processor. Even the "parallel" tests are actually serial simulations of a parallel run, in which we use a different random number stream for each lattice site. We have implemented the Metropolis and the Wolff algorithms. These tests take 6 command line arguments:

```
seed param lattice_size block_size discard_blocks use_blocks
```

For example:

```
mpirun -np 1 metropolis.lcg64 111 0 16 10 10 100
```

runs the Metropolis algorithm test using the 64 bit Linear Congruential Generator. The seed is 111, the parameter to the generator is 0 (which indicates the default multiplier for the LCG). We use a 16×16 lattice. The simulations starts with a random initial configuration. The simulation then performs several sweeps through the entire lattice. We record results of blocks of 10 sweeps. We discard the results of the first 10 blocks (thus a total of 100 sweeps) and then use the results of the next 100 blocks. We recommend that at least the first 1000 sweeps be discarded, though this number can be smaller for the Wolff algorithm. (In the actual RNG tests, we discarded the first 100000 sweeps, so that the initial configuration would be "forgotten".) The last parameter must be a power of 10. The tests are carried out with $J/K_bT = 0.4406868$, where J is the energy per bond and T is the temperature. This parameter can be changed in the code, to run the tests at a different temperature. Sample results are shown below:

```
Metropolis Algorithm with Serial PRNGs
lattice_size = 16, block_size = 10, discard_blocks = 10,
   use_blocks = 100

64 bit Linear Congruential Generator with Prime Additive Constant

seed = 111, stream_number = 0   parameter = 0

Streams are synchronized!
    Energy       Energy_error  Sigma_Energy  ...
0.  -1.4737500  0.0206851     0.0465998     ...
1.  -1.4819062  0.0288413     0.0141733     ...
```

The important field are the *Energy_error* which gives the observed error in Energy, and the *Sigma_Energy* which gives the standard error for the energy. We have similar fields for the specific heat, Cv. The first line is obtained after 10 blocks of data have been averaged. Each subsequent line is the average of ten times the previous run.

(2) *Random walk tests:* In the Random Walk test, we start a "Random Walker" from a certain position on a two dimensional lattice. The random walker then takes a certain number of steps to other lattice points. The direction of each step is determined from the value returned by a random number generated. A series of such tests are performed, and for each such test, the final position is noted, [Vattulainen et al. 1994].
The first 6 command line arguments for this test are as for the statistical tests. The seventh command line argument is the length of each walk. For example:

```
mpirun -np 2 random_walk.lcg64 2 3 0 0 10 0 100
```

runs the (parallel) random walk test using the 64 bit Linear Congruential Generator, where we interleave 3 streams at a time to produce a new stream; two such streams being tested. The seed is 0, the parameter to the generator is 0 (which indicated the default multiplier for the LCG). We perform 10 tests per stream. Each test uses a walk length of 100.

4.3 Summary of Test Results

We shall now give a summary of the tests results, since this is a major factor in the quality of the random number software. All the SPRNG generators were tested with the DIEHARD [Marsaglia] suite, and they all passed these tests, except that the lower order bits of the 48-bit LCG are not very random, as expected. The tests from Knuth [Knuth 1998] mentioned above were performed, including their parallel versions. At least around 10^{11} random numbers were tested for each generator in each test. The collisions test used 10^{12} random numbers. The serial gap test for the ALFG used 10^{13} random numbers – the largest empirical random number test ever accomplished. The parallel gap test for this generator used 10^{12} random numbers. (ALFGs are susceptible to the gap test, hence the larger tests for this generator.) We also performed the blocking test for parallel generators, and the Metropolis and Wolff algorithm for the Ising model with at least 10^{11} random numbers. More details can be found at our web site.

5. CONCLUSIONS AND THE FUTURE OF SPRNG

We have presented a considerable amount of detail about parallel pseudorandom number generation through parameterization. In particular, we have described the SPRNG library as an example of a comprehensive library for parallel Monte Carlo based on parameterized PRNGs. In addition, we have described in detail the suite of randomness tests that is part of the SPRNG library.

While care has been taken in constructing generators for the SPRNG package, the designers realize that there is no such thing as a PRNG that behaves flawlessly for every application. This is even more true when one considers using scalable platforms for Monte Carlo. The underlying recursions that are used for PRNGs are simple, and so they inevitably have regular structure. This deterministic regularity permits analysis of the sequences and is the PRNG's Achilles heel. Thus any large Monte Carlo calculation must be viewed with suspicion. There is always the possibility that a numerical result produced is incorrect due to an unfortunate interplay between the application and the PRNG used. The only way to prevent this is to treat each new Monte Carlo derived result as an experiment that must be controlled. The tools required to control problems with the PRNG include the ability to use another PRNG in the same calculation. In addition, one must be able to develop and use entirely new PRNGs as well. These capabilities, as well as parallel and serial tests of randomness, Cuccaro *et al.* [Cuccaro et al. 1995; Srinivasan et al. 1999b], are components that make the SPRNG package unique among tools for parallel Monte Carlo.

In the future, SPRNG will be extended in several ways. First, the **A**ccelerated **S**trategic **C**omputing **I**nitiative (ASCI) has begun support for further SPRNG development. This work will be aimed at adapting the existing SPRNG generators and finding new generators for SPRNG to be more appropriate for ASCI-class Monte Carlo applications that will run on the current and future generations of ASCI hardware. Among the generators that are planned for incorporation into SPRNG are the previously mentioned "Mersenne Twister" SRG and both the implicit and explicit inversive congruential generator, [Eichenauer and Lehn 1986; Niederreiter 1994; Hellekalek 2000]. Besides the search for more ASCI-appropriate generators, SPRNG will also be generalized to include explicit support for distributed computing on heterogeneous systems by incorporating SPRNG into the CONDOR, [Litzkow et al. 1998; Zhou and Mascagni 1999], distributed computing environment. The ability to use SPRNG on distributed systems will be used to provide computing cycles for several web-based applications involving SPRNG. These include the development of a web-based interface to a larger and more comprehensive testing suite and a Monte

Carlo numerical integration server. In addition, the design of SPRNG will be used as a model for the construction of a library to provide quasirandom numbers for parallel and distributed systems. Quasirandom numbers [Niederreiter 1988] are very uniform in their metrical distribution and hence are more effective at error reduction on certain Monte Carlo applications (such as numerical integration) than are pseudorandom numbers. However, while pseudorandom numbers are designed to pass tests of randomness, quasirandom numbers are often highly correlated by construction and thus fail these same randomness tests. None the less, the list of applications that benefit from the use of quasirandom numbers is sufficient in size to justify the efforts to provide software for their use on new architectures. Finally, SPRNG will be the random number server in a planned **P**roblem **S**olving **E**nvironment (PSE) for Monte Carlo computations based on the evaluation of path integrals on a wide range of architectures. Such a PSE for path integrals would have a wide range of applicability from particle physics to the pricing of financial derivatives.

ACKNOWLEDGMENTS

The first author wants to acknowledge the support and collaboration of Steven Cuccaro, Daniel Pryor, and Michael Robinson at the Institute for Defense Analyses' Center for Computing Sciences on the design, analysis, and implementation of the parallel ALFG code that served as the starting point for SPRNG.

REFERENCES

BEALE, P. 1996. Exact distribution of energies in the two-dimensional Ising model. *Phys. Rev. Lett. 76*, 78.

BRENT, R. P. 1992. Uniform random number generators for supercomputers. In *Proceedings Fifth Australian Supercomputer Conference* (1992), pp. 95–104. 5th ASC Organizing Committee.

BRENT, R. P. 1994. On the periods of generalized Fibonacci recurrences. *Mathematics of Computation 63*, 389–401.

BRILLHART, J., LEHMER, D. H., SELFRIDGE, J. L., TUCKERMAN, B., AND WAGSTAFF JR., S. S. 1988. *Factorizations of* $b^n \pm 1$ $b = 2, 3, 5, 7, 10, 11, 12$ *up to high powers.* Contemporary Mathematics, Volume 22, Second Edition. American Mathematical Society, Providence, Rhode Island.

CODDINGTON, P. 1994. Analysis of random number generators using Monte Carlo simulation. *Int. J. of Mod. Phys. C 5*, 3, 547–560.

CODDINGTON, P. 1996. Tests of random number generators using Ising model simulations. *Int. J. of Mod. Phys. "C" 7*, 3, 295–303.

CODDINGTON, P. 1997. Random number generators for parallel computers. http://nhse.cs.rice.edu/NHSEreview/96-2.html.

CUCCARO, S. A., MASCAGNI, M., AND PRYOR, D. V. 1995. Techniques for testing the quality of parallel pseudorandom number generators. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing* (Philadelphia, Pennsylvania, 1995), pp. 279–284. SIAM.

DEÁK, I. 1990. Uniform random number generators for parallel computers. *Parallel Computing 15*, 155–164.

DELEGLISE, M. AND RIVAT, J. 1996. Computing $\pi(x)$: the Meissel, Lehmer, Lagarias, Miller, Odlyzko method. *Math. Comput. 65*, 235–245.

DE MATTEIS, A. AND PAGNUTTI, S. 1990a. A class of parallel random number generators. *Parallel Computing 13*, 193–198.

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

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

De Matteis, A. and Pagnutti, S. 1995. Controlling correlations in parallel Monte Carlo. *Parallel Computing 21*, 73–84.

Eichenauer, J. and Lehn, J. 1986. A nonlinear congruential pseudorandom number generator. *Statist. Hefte 37*, 315–326.

Entacher, K. 1998. Bad subsequences of well-known linear congruential pseudorandom number generators. *ACM Trans. Model. Comput. Simul. 8*, 61–70.

Ferrenberg, A. M., Landau, D. P., and Wong, Y. J. 1992. Monte Carlo simulations: Hidden errors from "good" random number generators. *Phys. Rev. Let. 69*, 3382–3384.

Frederickson, P., Hiromoto, R., Jordan, T. L., Smith, B., and Warnock, T. 1984. Pseudo-random trees in Monte Carlo. *Parallel Computing 1*, 175–180.

Golomb, S. W. 1982. *Shift Register Sequences.* Aegean Park Press, Laguna Hills, California. Revised Edition.

Grassberger, P. 1993. On correlations in 'good' random number generators. *Phys. Lett. A 181*, 1, 43–46.

Hellekalek, P., et al. 2000. plab group home page. http://random.mat.sgb.ac.at.

Knuth, D. E. 1998. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Third edition.* Addison-Wesley, Reading, Massachusetts.

Kuipers, L. and Niederreiter, H. 1974. *Uniform distribution of sequences.* John Wiley and Sons, New York.

Lagarias, J. C., Miller, V. S., and Odlyzko, A. M. 1985. Computing $\pi(x)$: The Meissel-Lehmer method. *Mathematics of Computation 55*, 537–560.

L'Ecuyer, P. 1990. Random numbers for simulation. *Communications of the ACM 33*, 85–97.

L'Ecuyer, P. 1994. Uniform random number generation. *Ann. Oper. Res. 53*, 77–120.

L'Ecuyer, P. 1998. Random number generation. In J. Banks Ed., *Handbook of Simulation*, pp. 93–137. New York: Wiley.

L'Ecuyer, P. and Côté, S. 1991. Implementing a random number package with splitting facilities. *ACM Trans. on Math. Soft. 17*, 98–111.

Lehmer, D. H. 1949. Mathematical methods in large-scale computing units. In *Proceedings of the 2nd Symposium on Large-Scale Digital Calculating Machinery* (Cambridge, Massachusetts, 1949), pp. 141–146. Harvard University Press.

Lewis, T. G. and Payne, W. H. 1973. Generalized feedback shift register pseudorandom number algorithms. *Journal of the ACM 20*, 456–468.

Lidl, R. and Niederreiter, H. 1986. *Introduction to finite fields and their applications.* Cambridge University Press, Cambridge, London, New York.

Litzkow, M., Livny, M., and Mutka, M. W. 1998. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems* (June 1998), pp. 104–111.

Makino, J. 1994. Lagged-Fibonacci random number generator on parallel computers. *Parallel Computing 20*, 1357–1367.

Marsaglia, G. Diehard software package. ftp://stat.fsu.edu/pub/diehard.

Marsaglia, G. 1968. Random numbers fall mainly in the planes. *Proc. Nat. Acad. Sci. U.S.A. 62*, 25–28.

Marsaglia, G. 1972. The structure of linear congruential sequences. In S. K. Zaremba Ed., *Applications of Number Theory to Numerical Analysis*, pp. 249–285. New York: Academic Press.

Marsaglia, G. 1985. A current view of random number generators. In *Computing Science and Statistics: Proceedings of the XVIth Symposium on the Interface* (1985), pp. 3–10.

Marsaglia, G. and Tsay, L.-H. 1985. Matrices and the structure of random number sequences. *Linear Alg. and Applic. 67*, 147–156.

Mascagni, M. 1997. A parallel non-linear Fibonacci pseudorandom number generator. Abstract, 45th SIAM Annual Meeting.

Mascagni, M. 1998. Parallel linear congruential generators with prime moduli. *Parallel Computing (and 1997 IMA Preprint #1470) 24*, 923–936.

Mascagni, M. 1999a. Serial and parallel random number generation. In P. Nightingale and C. Umrigar Eds., *Quantum Monte Carlo in Physics and Chemistry*, NATO ASI Series, Series C: Mathematical and Physical Sciences, pp. 425–446. Dordrecht: Kluwer.

Mascagni, M. 1999b. Some methods of parallel pseudorandom number generation. In R. Schreiber, M. Heath, and A. Ranade Eds., *Proceedings of the IMA Workshop on Algorithms for Parallel Processing* (1999), pp. 277–288. Springer-Verlag: New York, Berlin.

Mascagni, M., Cuccaro, S. A., Pryor, D. V., and Robinson, M. L. 1995. A fast, high-quality, and reproducible lagged-Fibonacci pseudorandom number generator. *J. Comput. Physics 15*, 211–219.

Mascagni, M., Robinson, M. L., Pryor, D. V., and Cuccaro, S. A. 1995. Parallel pseudorandom number generation using additive lagged-Fibonacci recursions. *Springer Verlag Lecture Notes in Statistics 106*, 263–277.

Massey, J. L. 1969. Shift-register synthesis and BCH decoding. *IEEE Trans. Information Theory IT-15*, 122–127.

Matsumoto, M. and Kurita, Y. 1992. Twisted GFSR generators. *ACM Trans. Model. Comput. Simul. 2*, 179–194.

Matsumoto, M. and Nishimura, T. 1998. Mersenne twister: A 623-dimensionally equidistributed uniformd pseudo-random number generators. *ACM Trans. Model. Comput. Simul. 8*, 3–30.

Niederreiter, H. 1988. Low-discrepancy and low-dispersion sequences. *J. Number Theory 30*, 51–70.

Niederreiter, H. 1992. *Random number generation and quasi-Monte Carlo methods.* SIAM, Philadelphia, Pennsylvania.

Niederreiter, H. 1994. On a new class of pseudorandom numbers for simulation methods. *J. Comput. Appl. Math. 56*, 159–167.

Park, S. K. and Miller, K. W. 1988. Random number generators: good ones are hard to find. *Communications of the ACM 31*, 1192–1201.

Percus, O. E. and Kalos, M. H. 1989. Random number generators for MIMD parallel processors. *J. of Par. Distr. Comput. 6*, 477–497.

Pryor, D. V., Cuccaro, S. A., Mascagni, M., and Robinson, M. L. 1994. Implementation and usage of a portable and reproducible parallel pseudorandom number generator. In *Proceedings of Supercomputing '94* (1994), pp. 311–319. IEEE.

Schmidt, W. 1976. Equations over finite fields: An elementary approach. In *Lecture Notes in Mathematics #536*. Berlin, Heidelberg, New York: Springer-Verlag.

Selke, W., Talapov, A. L., and Schur, L. N. 1993. Cluster-flipping Monte Carlo algorithm and correlations in "good" random number generators. *JETP Lett. 58*, 8, 665–668.

Srinivasan, A., Ceperley, D., and Mascagni, M. 1999a. Random number generators for parallel applications. In D. Ferguson, J. I. Siepmann, and D. G. Truhlar Eds., *Monte Carlo Methods in Chemical Physics*, Volume 105 of *Advances in Chemical Physics*, pp. 13–36. New York: John Wiley and Sons.

Srinivasan, A., Ceperley, D., and Mascagni, M. 1999b. Testing parallel random number generators. Submitted.

Tausworthe, R. C. 1965. Random numbers generated by linear recurrence modulo two. *Math. Comput. 19*, 201–209.

Tezuka, S. 1995. *Uniform Random Numbers: Theory and Practice*. Kluwer Academic Publishers, Norwell, Massachusetts.

Vattulainen, I. 1999. Framework for testing random numbers in parallel calculations. *Phys. Rev. E 59*, 7200–7204.

Vattulainen, I., Ala-Nissila, T., and Kankaala, K. 1994. Physical tests for random numbers in simulations. *Phys. Rev. Lett. 73*, 2513–2516.

Zhou, M. and Mascagni, M. 1999. Parallel Monte Carlo in a distributed environment: `SPRNG` and `condor`. In *to appear in Proceedings of the First Southern Symposium on Computing* (1999).