Theory and Software for Parallel Random Number Generation

Michael Mascagni
Department of Computer Science
203 Love Building
Florida State Univeristy
Tallahassee, FL 32306-4503, USA
mascagni@cs.fsu.edu

Abstract

In this article we outline some methods for parallel pseudorandom number generation. In addition, we present some rationale for choosing good parallel pseudorandom generators. We begin with methods based on splitting the full-period sequences of conventional pseudorandom number generators into subsequences. Given the subsequences, each parallel process requiring random numbers is then given a different subsequence for this purpose. The splitting methods considered include breaking the original sequence into nonoverlapping subsequence blocks, using the leapfrog technique to make subsequences, using recursive-halving leap ahead for subsequence definition, and using the Lehmer tree for subsequence construction. Unfortunately, there are some generic reasons why using a single full-period generator to provide parallel subsequences is a dangerous procedure, and so we then present an alternative: using parameterized, full-period pseudorandom number sequences to provided parallel random numbers. We then present several methods of parallel pseudorandom generation based on parameterization. We describe parameterized versions of the following common 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. Finally, we present a short description of a scalable library for pseudorandom number generation, called SPRNG. SPRNG is based on associating each parallel process which requires random numbers with a unique, parameterized, full-period pseudorandom number generator. This is generically accomplished by mapping the parameterized generators onto a binary tree so that new generators may be initialized without any interprocessor communication and that no parallel processes ever use the same parameterized generator. The brief description of SPRNG contained within this document is meant only to outline the rationale behind and the capabilities of SPRNG. Much more information, including the library's source code, usage examples, and detailed documentation aimed at helping users with putting and using SPRNG on scalable and distributed parallel systems is available at the URL: http://sprng.cs.fsu.edu.

1 Introduction

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

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

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.
We briefly present several methods for parallel pseudorandom number generation and discuss pros and cons for each method. If the reader is interested in background material on plain old serial pseudo-random number generation in general, consult the following references by Knuth [12], L'Ecuyer [15], Niederreiter [36], and Park and Miller [37], while a good overview of parallel pseudorandom number generation can be found in a recent work by the present article's author [26, 29, 41].
In our parallel pseudorandom number generation review we are interested, primarily, with methods for obtaining **p**arallel **p**seudo**r**andom **n**umber **g**enerators (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 also briefly consider the production of parallel streams of pseudorandom numbers by taking substreams from a single, long-period PRNG. However, we will state several, generic, reasons why PPRNGs constructed via splitting are ill advised, and so our focus will again return to parameterization. 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.
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. However, in §2.1 we will present, for purposes of completeness, an overview of methods based on splitting single PRNGs into substreams for use in parallel settings. In §2.2 two methods for parameterizing **l**inear **c**ongruential **g**enerators (LCGs). In §2.3 we present a parameterization of another linear method: **s**hift-**r**egister **g**enerators (SRGs). This parameterization is analogous to one of the LCG parameterizations presented in §2.2. In §2.4 we consider the parallel parameterization of so-called lagged-Fibonacci generators. In §3, we present the **S**calable **P**arallel **R**andom **N**umber **G**enerators (SPRNG) library, a comprehensive tool for parallel and distributed pseudorandom number generation developed by the authors. Finally in §4 we discuss open problems, and provide concluding remarks.

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; however, we begin with a review of methods for splitting full-period generators into subsequences for use as PPRNGs.

2.1 Spitting Techniques

Let us assume that we have a PRNG that generates the numbers $x_0, x_1, \ldots,$ and has period equal to $Per(x_i)$. In addition, let us assume that it is easy to leap ahead in this PRNG sequence an arbitrary amount in a manner that is computationally effective. With these constraints one can understand the two most common techniques for splitting a long serial PRNG into parallel subsequences: the blocking and the leap-frog techniques. In blocking, we associate a single block of length L with each subsequence, thus the first block will be $\{x_0, x_1, \ldots, x_{L-1}\}$, the second $\{x_L, x_{L+1}, \ldots, x_{2L-1}\}$, and the ith $\{x_{(i-1)L}, x_{(i-1)L+1}, \ldots, x_{iL-1}\}$. The other common technique, the leaf frog method, produces subsequences of length L as follows. First we define the leap ahead of $\ell = \lfloor \frac{Per(x_i)}{L} \rfloor$. Then the first leap frogged subsequence is $\{x_0, x_\ell, x_{2\ell}, \ldots, x_{(L-1)\ell}\}$. Similarly, the second leap-frogged

subsequence is $\{x_1, x_{1+\ell}, x_{1+2\ell}, \ldots, x_{1+(L-1)\ell}\}$, and the ith leap-frogged sequence is thus given by $\{x_i, x_{i+\ell}, x_{i+2\ell}, \ldots, x_{i+(L-1)\ell}\}$.
In the blocking approach, the ith element of successive subsequences are all exactly L apart, while successive elements in the leap-frogged subsequences are exactly ℓ apart with respect to the underlying PRNG. It is well known, that in most random number generators, strong correlations exist between numbers in the sequence separated by a constant amount, [7, 8, 9]. Thus, it is felt by many that these simple splitting techniques are doomed to suffer from correlations and hence to perform less well than expected in parallel. One solution, it would seem, is to consider splitting techniques where substreams are not constructed from the serial PRNG with constant offsets in either inter- or intrastream situations. Below, we outline two such techniques, the "Lehmer Tree" and the "Recursive Halving Leap-Ahead" method.

2.1.1 The Lehmer Tree

We begin this discussion by reviewing a splitting scheme called the "Lehmer tree" originally proposed as a method for splitting linear congruential generators for asynchronous, parallel machines, [10]. The idea is to choose two linear congruential generators, called the right and the left generators, to generate a tree of pseudorandom numbers. The motivation for a tree-structured stream of pseudorandom numbers is to ensure reproducible Monte Carlo computations on a distributed memory parallel computer.
Most agree that any PRNG must give reproducible output in the sense that the same seeding of a generator must produce the same stream of pseudorandom numbers. Any deterministic recursive sequence is reproducible in this sense when implemented in serial. In parallel, reproducibility can be assured only by an implementation that is insensitive to temporal relationships among processors. Since to be reproducible, the random numbers must be completely determined by which the parallel process they are generated within, not the physical processor on which the generator executes. A general way to assure this to employ a deterministic algorithm for seeding or parameterizing new random number generation processes that depends only on information known by the parent processes. One can view this requirement as leading to a logical tree of random number generators.
In the Lehmer tree, one generator (the right) is used to generate random numbers within a parallel process while the other (left) generator is used to seed generators on derivative parallel processes. In their analysis of the relationship between the right and left generators, Frederickson, et al., [10], state that (in their notation) $L(x) = R^W(x)$ holds for some $W > 0$. Here $L(x)$ is the left generator applied to x, and $R^W(x)$ is the result of W applications of the right generator to x. The left generator is equivalent to leaping ahead W elements in the right sequence to seed the next generators. One of the many elegant concepts in the original presentation of the Lehmer tree was that the left generator could be represented as a fixed power of the right generator. It so happens that any leap ahead, W, of a linear congruential generator can be expressed as a single step of another linear congruential generator.[2] Thus for linear congruential generators, the concept of the Lehmer tree can be thought of as using a precomputed leap-ahead value to split the sequence into a fixed number of subsequences.

2.1.2 Recursive Halving Leap-Ahead

We now consider how to extend the concept of the Lehmer tree to more general splittings. An undesirable property of the Lehmer tree is that the left generator, or equivalently the leap ahead, W, is fixed. This means that splitting a previously split sequence can lead to considerable overlap with

[2]Let the right generator be $R(x) = ax + b \pmod{m}$. $L(x) = R^W(x) = a^W x_0 + b(a^{W-1} + a^{W-2} + \cdots + a + 1) \pmod{m}$. With $A = a^W \pmod{m}$ and $B = b(a^{W-1} + a^{W-2} + \cdots + a + 1) \pmod{m}$, $L(x) = Ax + B \pmod{m}$.

the grandparent sequence. To remedy this problem, one can use a splitting that separates maximally from the original stream while also preventing overlap with all previously split streams. This can be accomplished by using variable sized leap-ahead values.
As above, assume the pseudorandom sequence $x_0, x_1, \ldots$ has period equal to $Per(x_i)$, and has already been split l times. In the next splitting we leap ahead $\lfloor \frac{Per(x_i)}{2^{l+1}} \rfloor$ in the sequence. For book keeping purposes we think of the original and the split sequence as having been split $(l+1)$ times. Thus the first split leaps ahead about one half the period, the next about one quarter, and so on. We call this procedure recursive halving leap-ahead.

2.1.3 Concluding Remarks on Splitting

When considering splitting a single sequence for parallel use, one must keep in mind that the parent sequence must have a period of sufficient length to provide all the random numbers required for all the parallel processes. Thus, the period of parent generators will have to be very very big. Unfortunately, for most generators a longer period implies that the cost of generating each random number goes up. Thus, for generators of this type, which include most commonly used generators, splitting is not scalable in this parallel context. In addition, there is often correlation in the parent sequence that is quite deleterious for split parallel implementation. Thus, we feel that splitting is not an acceptable way to parallelize PRNGs, except with certain types of generators. Thus, below we begin the development of the notion of parameterization as a means of parallelizing PRNGs. However, for readers interested in splitting methods and the consequences of using split streams in parallel please consult the works by Deák [5], De Matteis and Pagnutti [7, 8, 9], Frederickson *et al.* [10], and L'Ecuyer and Côté [16].

2.2 *Linear Congruential Generators*

The most commonly used generator for pseudorandom numbers is the LCG. The LCG was first proposed for use by Lehmer [17], 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 [22]. 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 [28], and leads to several interesting computational problems.

2.2.1 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 [12].[3] Given primitivity, one can use

[3]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 $\{1 \le i \le$

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.* [3], efficient algorithms for computing ℓ_j can be found in a recent work of the author [28]. 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, $\{x_n\}_{n=0}^{k-1}$, as:

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

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 [13]. 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:

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

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.* [39]. 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 [40]:

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

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 [28]; however, two important open questions remain: (i) 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 (ii) does the good interstream correlation of Eq. (4) also ensure good intrastream independence via the spectral test?

$m - 1\}$.

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.

2.2.2 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 [38], 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 [38]. 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 [6], Lagarias, Miller, and Odlyzko [14] . 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 [28]. It is believed that the issues for computing $\pi^{-1}(j)$ are similar.
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 [12]. 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 §2.2.1.

2.3 Shift-Register Generators

Shift register generators (SRGs) are linear recursions modulo 2, see Golomb [11], Lewis and Payne [18], and Tausworthe [42], 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, see Lidl and Niederreiter [19]:

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

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 §2.2.1. 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, see Lidl and Niederreiter [19]. 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, see Massey [32] 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, see Niederreiter [36]. 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.

2.4 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 additive lagged-Fibonacci generator (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, see Makino [21]. 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: (i) it is easy to implement, (ii) it is cheap to compute using Eq. (7), and (iii) the ALFG does well on standard statistical tests, see Marsaglia [24].
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 [2] and Marsagia and Tsay [25], from which one can infer that this generator has $2^{(k-1)\times(m-1)}$ different full-period cycles, Mascagni *et al.* [30]. 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, $\{x_0, x_1, \dots, x_{k-1}\}$, can be bit-wise initialized using the following template:

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

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.* [30]. 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:

$$\begin{array}{cccc|c|c}
\text{m.s.b.} & & & & \text{l.s.b.} & \\
b_{m-1} & b_{m-2} & \dots & b_1 & b_0 & \\
\hline
\blacksquare & \blacksquare & \dots & \blacksquare & b_{0k-1} & x_{k-1} \\
\blacksquare & \blacksquare & \dots & \blacksquare & b_{0k-2} & x_{k-2} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \\
\blacksquare & \blacksquare & \dots & \blacksquare & b_{01} & x_1 \\
\hline
0 & 0 & \dots & 0 & 1 & x_0
\end{array} \tag{9}$$

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.* [31]. However, it is easy to avoid all but the most uncorrelated pairs in a computation, Pryor *et al.* [39]. 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.* [31], 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 [1].

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 [25], it has empirical properties considered to be superior to ALFGs, Marsaglia [24]. Of interest for parallel computing is that a parameterization analogous to that of the ALFG exists for the MLFG, see Mascagni [27].

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.[4] SPRNG includes the following:

- Several, qualitatively distinct, well tested, scalable RNGs
- 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

[4]It is important to note, that while SPRNG currently *only* has parameterized generators available, future, short-term, plans include the inclusion of the Mersenne Twister generator of Matsumoto and collaborators, [33, 34]. The Mersenne Twister is a particularly efficient approach to implementing a SRG, and provides an extraordinarily long period. With this generator, we plan to parameterize via pseudorandom seeding.

The decision to use parameterized generators was based on work of the author in parameterizing several different, common, RNGs 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: $x_n = ax_{n-1} \pmod{m}$
- Power-of-two modulus linear congruential: $x_n = ax_{n-1} + b \pmod{2^m}$
- 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 $y_n = 107374182y_{n-1} + 104480y_{n-5} \pmod{2147483647}$

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 RNGs 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 RNG by rewriting two dummy SPRNG two 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.[5] When a different RNG is desired, e.g. when a particular RNG is thought to give spurious results in a given application, a qualitatively different generator can replace the original by merely relinking the user program with SPRNG.[6] Finally, new RNGs can be incorporated into SPRNG with little more than coding the generation and initialization routines and recompiling SPRNG.

4 Conclusions and Open Problems

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.
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 are 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 as an unfortunate interplay between the application and PRNG may result in spurious results. 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.* [4], are components that make the SPRNG package unique among tools for parallel Monte Carlo.

Acknowledgement

SPRNG was originally developed with funding from the U.S. Defense Advanced Research Projects Administration (DARPA) through a collaboration between the author's group and the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign. Currently, SPRNG research and development is supported through funding of the author and his group in the Computer Science Department at The Florida State University by the U.S. Department of Energy (DOE) though it's Accelerated Strategic Computing Initiative (ASCI). Additional funding for this work comes from the U.S. National Science Foundation (NSF) and the U.S. Army Research Office (ARO).

References

[1] R. P. BRENT, "Uniform Random Number Generators for Supercomputers" in *Proceedings Fifth Australian Supercomputer Conference*, 5th ASC Organizing Committee, pp. 95–104, 1992.

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

[5]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.* [20].

[6]In version 2.0 of SPRNG one can obtain all the different SPRNG generators directly via a single library and name space.

[3] J. BRILLHART, D. H. LEHMER, J. L. SELFRIDGE, B. TUCKERMAN AND S. S. WAGSTAFF, JR., "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, 1988.

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

[5] I. DEÁK, "Uniform random number generators for parallel computers," *Parallel Computing*, **15**: 155–164, 1990.

[6] M. DELEGLISE AND J. RIVAT, "Computing $\pi(x)$: the Meissel, Lehmer, Lagarias, Miller, Odlyzko method," *Mathematics of Computation*, **65**: 235–245, 1996.

[7] A. DE MATTEIS AND S. PAGNUTTI, "Parallelization of random number generators and long-range correlations," *Parallel Computing*, **15**: 155–164, 1990.

[8] A. DE MATTEIS AND S. PAGNUTTI, "A class of parallel random number generators," *Parallel Computing*, **13**: 193–198, 1990.

[9] A. DE MATTEIS AND S. PAGNUTTI, "Long-range correlations in linear and non-linear random number generators," *Parallel Computing*, **14**: 207–210, 1990.

[10] P. FREDERICKSON, R. HIROMOTO, T. L. JORDAN, B. SMITH AND T. WARNOCK, "Pseudo-random trees in Monte Carlo," *Parallel Computing*, **1**: 175–180, 1984.

[11] S. W. GOLOMB, *Shift Register Sequences*, Revised Edition, Aegean Park Press, Laguna Hills, California, 1982.

[12] D. E. KNUTH, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*, Third Edition, Addison-Wesley: Reading, MA, 1998.

[13] L. KUIPERS AND H. NIEDERREITER, *Uniform distribution of sequences*, John Wiley and Sons: New York, 1974.

[14] J. C. LAGARIAS, V. S. MILLER AND A. M. ODLYZKO, "Computing $\pi(x)$: The Meissel-Lehmer method," *Mathematics of Computation*, **55**: 537–560, 1985.

[15] P. L'ECUYER, "Random numbers for simulation," *Communications of the ACM*, **33**: 85–97, 1990.

[16] P. L'ECUYER AND S. CÔTÉ, "Implementing a random number package with splitting facilities," *ACM Trans. on Mathematical Software*, **17**: 98–111, 1991.

[17] D. H. LEHMER, "Mathematical methods in large-scale computing units," in *Proc. 2nd Symposium on LargeScale Digital Calculating Machinery*, Harvard University Press: Cambridge, Massachusetts, pp. 141–146, 1949.

[18] T. G. LEWIS AND W. H. PAYNE, "Generalized feedback shift register pseudorandom number algorithms," *Journal of the ACM*, **20**: 456–468, 1973.

[19] R. LIDL AND H. NIEDERREITER, *Introduction to finite fields and their applications*, Cambridge University Press: Cambridge, London, New York, 1986.

[20] M. LITZKOW, M. LIVNY, AND M. W. MUTKA, "Condor - A Hunter of Idle Workstations," *Proceedings of the 8th International Conference of Distributed Computing Systems*, pp. 104–111, June, 1988.

[21] J. MAKINO, "Lagged-Fibonacci random number generator on parallel computers,"*Parallel Computing*, **20**: 1357–1367, 1994.

[22] G. MARSAGLIA, "Random numbers fall mainly in the planes," *Proc. Nat. Acad. Sci. U.S.A.*, **62**: 25–28, 1968.

[23] G. MARSAGLIA, "The structure of linear congruential sequences," in *Applications of Number Theory to Numerical Analysis*, S. K. Zaremba, Ed., Academic Press, New York, pp. 249–285, 1972.

[24] G. MARSAGLIA, "A current view of random number generators," in *Computing Science and Statistics: Proceedings of the XVIth Symposium on the Interface*, pp. 3–10, 1985.

[25] G. MARSAGLIA AND L.-H. TSAY, "Matrices and the structure of random number sequences," *Linear Alg. and Applic.*, **67**: 147–156, 1985.

[26] M. MASCAGNI AND A. SRINIVASAN, "SPRNG: A Scalable Library for Random Number Gerneration," *ACM Transactions on Mathematical Software*, in the press, 2000.

[27] M. MASCAGNI, "A parallel non-linear Fibonacci pseudorandom number generator," abstract, 45th SIAM Annual Meeting, 1997.

[28] M. MASCAGNI, "Parallel linear congruential generators with prime moduli," *Parallel Computing*, **24**: 923-936, 1998 and 1997 IMA Preprint #1470.

[29] M. MASCAGNI, "Some methods of parallel pseudorandom number generation," to appear in *Proceedings of the IMA Workshop on Algorithms for Parallel Processing*, R. Schreiber, M. Heath and A. Ranade editors, Springer-Verlag: New York, Berlin, 1998.

[30] M. MASCAGNI, S. A. CUCCARO, D. V. PRYOR AND M. L. ROBINSON, "A fast, high-quality, and reproducible lagged-Fibonacci pseudorandom number generator," *Journal of Computational Physics*, **15**: 211–219, 1995.

[31] M. MASCAGNI, M. L. ROBINSON, D. V. PRYOR AND S. A. CUCCARO, "Parallel pseudorandom number generation using additive lagged-Fibonacci recursions," *Springer Verlag Lecture Notes in Statistics*, **106**: 263–277, 1995.

[32] J. L. MASSEY, "Shift-register synthesis and BCH decoding," *IEEE Trans. Information Theory*, **IT-15**: 122–127, 1969.

[33] M. MATSUMOTO AND Y. KURITA, "Twisted GFSR generators," *ACM Trans. Model. Comput. Simul.*, **2**: 179–194, 1992.

[34] M. MATSUMOTO AND T. NISHIMURA, "Mersenne twister: A 623-dimensionally equidistributed uniformd pseudo-random number generators," *ACM Trans. Model. Comput. Simul.*, **8**: 3–30, 1998.

[35] H. NIEDERREITER, "Low-discrepancy and low-dispersion sequences," *J. Number Theory*, **30**: 51–70, 1988.

[36] H. NIEDERREITER, *Random number generation and quasi-Monte Carlo methods*, SIAM: Philadelphia, Pennsylvania, 1992.

[37] S. K. PARK AND K. W. MILLER, "Random number generators: good ones are hard to find," *Communications of the ACM*, **31**: 1192–1201, 1998.

[38] O. E. PERCUS AND M. H. KALOS, "Random number generators for MIMD parallel processors," *J. of Par. Distr. Comput.*, **6**: 477–497, 1989.

[39] D. V. PRYOR, S. A. CUCCARO, M. MASCAGNI AND M. L. ROBINSON, "Implementation and usage of a portable and reproducible parallel pseudorandom number generator," in *Proceedings of Supercomputing '94*, IEEE, pp. 311–319, 1994.

[40] W. SCHMIDT, *Equations over Finite Fields: An Elementary Approach*, Lecture Notes in Mathematics #536, Springer-Verlag: Berlin, Heidelberg, New York, 1976.

[41] A. SRINIVASAN, D. M. CEPERLEY AND M. MASCAGNI, "Random Number Generators for Parallel Applications," to appear in *Monte Carlo Methods in Chemical Physics*, D. Ferguson, J. I. Siepmann, and D. G. Truhlar, editors, Advances in Chemical Physics series, Volume 105, John Wiley and Sons, New York, to appear in 1998.

[42] R. C. TAUSWORTHE, "Random numbers generated by linear recurrence modulo two," *Mathematics of Computation*, **19**: 201–209, 1965.