from *SIAM News*, Volume 32, Number 5

Parallel Pseudorandom Number Generation

By Michael Mascagni

APPLICATIONS ON ADVANCED ARCHITECTURE COMPUTERS

Greg Astfalk, Editor

Monte Carlo applications are widely perceived as embarrassingly parallel. (Monte Carlo enthusiasts prefer the term "naturally parallel" to the somewhat derogatory "embarrassingly parallel" coined by computer scientists.) 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 the calculation in the same amount of time. This will be true only if the results in each processor are statistically independent, which will be true, in turn, only if the streams of random numbers generated in each processor are independent.

In this article, we outline some methods for *parallel* pseudorandom number generation. We consider parameterized versions of three random number generators. The exact meaning of parameterization depends on the type of parallel pseudorandom number generator (PPRNG) under discussion. (We distinguish parameterization from splitting methods. Splitting is the production of parallel streams of pseudorandom numbers by taking substreams from a single, long-period PRNG.) 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 for specifying valid parameter values and consider these choices in terms of the quality of the pseudorandom numbers produced.

Linear Congruential Generators

The most commonly used generator of pseudorandom numbers is the linear congruential generator (LCG). The LCG was first proposed for use by Lehmer (circa 1949) and is referred to as the Lehmer generator in the early literature. The linear recursion underlying LCGs is:

$$x_n = ax_{n-1} + b \pmod{m} \qquad (1)$$

When the multiplier, a, additive constant, b, and modulus, m, are chosen appropriately, we obtain 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 [8]. The x_n's in (1) are integer residues modulo m. 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 implementation reasons it is always chosen to be either prime or a power of two. The parameterization method used is based on the type of modulus that has been chosen. When m is prime, a method based on use of the multiplier a as the parameter has been proposed. This choice, the rationale for which is outlined in [12], leads to several interesting computational problems.

To parameterize a when m is prime, we must first determine the family of permissible a's. When m is prime and we want to obtain the maximal period (of length $m - 1$ in this case), a condition on a is that it be a primitive element modulo m [4]. An integer a is primitive modulo m if the set of integers $\{a^i \pmod{m} \mid 1 \le i \le m - 1\}$ equals the set $\{1 \le i \le m - 1\}$. Given primitivity, we can make use of the following fact: If a and α are primitive elements modulo m, then $\alpha = a^i \pmod{m}$ for some i relatively prime to $\phi(m)$. When m is prime, $\phi(m) = m - 1$. 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 $m - 1$, efficient algorithms for computing ℓ_j can be found [12].

Two important questions remain open: (1) Overall, is it more efficient to choose m to be amenable to fast modular multiplication or to fast calculation of the jth integer relatively prime to $m - 1$? (2) Does the good inter-stream correlation mentioned in [12] also ensure good intra-stream independence via the spectral test? The first question is of practical interest with respect to performance. For the second, however, a negative answer would mean that such techniques are less attractive for parallel pseudorandom number generation.

An alternative way to use LCGs to make a PPRNG is to parameterize the additive constant in (1) when the modulus is a power of two, i.e., when $m = 2^k$ for some integer $k > 1$. This technique was first proposed in [17] as a way to provide a PPRNG for the NYU Ultra-computer. The technique has some interesting advantages over parameterization of the multiplier. Use of power-of-two modulus LCGs also has considerable disadvantages, however.

The parameterization chooses a set of additive constants $\{b_j\}$ that are pair-wise 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 ensures the pairwise relative primality and also provides the largest set of such

residues. With this choice, certain favorable inter-stream properties can be theoretically derived from the spectral test [17]. However, the difficult problem of computing the jth prime arises. This aspect of the generator is not discussed in detail in [17], in part because the authors expect to provide only a small number of PRNGs. When the number of PRNGs to be provided with this method is large, fast algorithms can be used for the computation of $\pi(x)$, the number of primes less than x [3].This is the inverse of the desired function, so we designate $\pi^{-1}(j)$ as the jth prime. The details of such an implementation need to be specified, but a related method for computing the jth integer relatively prime to a given set of integers is given in [12]. It is believed that the issues for computing $\pi^{-1}(j)$ are similar.

One important advantage of this parameterization is an inter-stream correlation measure based on the spectral test suggesting that there will be good inter-stream independence. Given that the spectral test for LCGs measures the quality of the multiplier, this sort of result is to be expected. A disadvantage of this parameterization is the need to compute $\pi^{-1}(j)$ if a large number of streams are to be provided. Regardless of the efficiency of the implementation, this is known to be a difficult computation with respect to computational complexity.

Finally, one of the biggest disadvantages of a power-of-two modulus is that 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 $0 < j < k$ [4]. In particular, the least significant bit of the LCG will alternate between 0 and 1. This is such a major shortcoming that it motivated us to consider param-eterizations of prime modulus LCGs, as discussed earlier.

Shift-register Generators

Shift-register generators (SRGs) are linear recursions modulo 2 [5] of the form:

$$x_{n+k} = \sum_{i=0}^{k-1} a_i x_{n+i} \pmod 2 \qquad (2)$$

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 [6]:

$$f(x) = x^k + \sum_{i=0}^{k-1} a_i x^i \pmod 2 \qquad (3)$$

If we want 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 (2) is very cheap to evaluate. For this reason, known primitive trinomials are often used to specify SRG recursions. This leads to very efficient, two-term recursions.

There are two ways to make pseudorandom integers from the bits produced by (2). The first, called the digital multi-step method, takes successive bits from (2) to form an integer of the desired length. With the digital multistep method, n iterations of (2) are required to produce a new n-bit pseudorandom integer. The second method, called the generalized feedback SRG, creates a new n-bit pseudorandom integer for every iteration of (2). This is done by constructing the n-bit word from x_{n+k} and $n-1$ other bits from the k bits of the SRG state. Although these two methods seem different, they are closely related and theoretical results for one always hold for the other.

One technique for parameterizing SRGs is analogous to the LCG parameterization discussed in the preceding section. There we took the object that made the LCG full-period, the primitive root multiplier, and found a representation for all the multipliers. Analogously, 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 SRG are themselves maximal and unique with respect to cyclic reordering [6]. We seek to identify those decimations.

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 $2^k - 2/k$. 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 $2^k - 2/k$ 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 proceeds as follows: Assume that 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 to 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 [15] in time linear in k.

This parameterization is relatively efficient when the binary polynomial algebra is implemented correctly. Use of such a parameterization has one major drawback: Although the reference primitive polynomial $p(x)$ may be sparse, the new polynomials are not necessarily sparse. (A polynomial is considered sparse if most of the a_i's in (2) are zero.) The cost of stepping (2) once is proportional to the number of non-zero a_i's in (2). We can thus significantly increase the bit-operational complexity of an SRG in this manner.

The similarity of the parameterization methods for prime modulus LCGs and SRGs is no accident. Both are based on maximal-

period linear recursions over a finite field. Thus, the discrepancy and exponential sum results for the two types of generators are similar [16].

Lagged-Fibonacci Generators

The generators discussed in the preceding sections can be parallelized by variation of a parameter in the underlying recursion. The additive lagged-Fibonacci generator (ALFG) can be parameterized through its initial values. The ALFG can be written as:

$$x_n = x_{n-j} + x_{n-k} \pmod{2^m}, \quad j < k \quad (4)$$

In recent years the ALFG has become a popular generator for serial and scalable parallel machines. In fact, the ALFG with $j = 5$, $k = 17$, and $m = 32$ was the standard PPRNG in the Thinking Machines Connection Machine Scientific Subroutine Library. This generator has become popular for a variety of reasons: It is easy to implement, computation of (4) is cheap, and it does well on standard statistical tests [9].

An important property of the ALFG is that the maximal period is $(2^k - 1)\, 2^{m-1}$. This occurs in very specific circumstances [1, 10], from which we can infer that this generator has $2^{(k-1)\times(m-1)}$ different full-period cycles [13]. This means that the state space of the ALFG is toroidal, with (4) providing the algorithm for movement in one of the torus dimensions. It is clear that finding the algorithm for movement in the other dimension is the basis of a very interesting parameterization. Since (4) tells us how to cycle over the full period of the ALFG, we must find a seed that is not in a given full-period cycle in order to move in the second dimension. The key 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: We can prove that the initial seed, $\{x_0, x_1, \ldots x_{k-1}\}$, can be bit-wise initialized with the following template (m.s.b. is the most significant bit and l.s.b. the least significant bit):

m.s.b.				l.s.b.	
b_{m-1}	b_{m-2}	...	b_1	b_0	
■	■	...	0	0	x_{k-1}
0	■	...	■	0	x_{k-2}
⋮	⋮	⋱	⋮	⋮	
■	0	...	■	0	x_1
■	■	...	■	1	x_0

(5)

Each square is a bit location to be assigned. Each unique assignment gives a seed in a provably distinct full-period cycle [13]. The least significant bits, b_0, are specified to be a fixed, non-zero pattern. If we allow an $O(k^2)$ precomputation to find a particular least-significant-bit pattern, the template is particularly simple:

m.s.b.				l.s.b.	
b_{m-1}	b_{m-2}	...	b_1	b_0	
■	■	...	■	b_{0k-1}	x_{k-1}
■	■	...	■	b_{0k-2}	x_{k-2}
⋮	⋮	⋱	⋮	⋮	
■	■	...	■	b_{01}	x_1
0	0	...	0	b_{00}	x_0

(6)

The elegance of this explicit parameterization leads us to 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[14]. However, it is easy to avoid all but the most uncorrelated pairs in a computation. In this case, there is extensive empirical evidence that the full-period exponential sum correlation between streams is $O\left(((2^k - 1)2^{m-1})^{1/2}\right)$, 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 [14] is an important open problem in the theory of ALFGs.

Another advantage of the ALFG is that it can be implemented directly with floating-point numbers, thereby avoiding the constant conversion from integer to floating-point that accompanies the use of other generators. When only floating-point numbers are required in the Monte Carlo computation, a distinct improvement in speed is achieved. To ensure that the parallel streams remain unique, however, care must be taken to maintain the identity of the corresponding integer recursion when the floating-point ALFG is used in parallel.

The multiplicative lagged-Fibonacci generator (MLFG) is an interesting cousin of the ALFG. It is defined by:

$$x_n = x_{n-j} \times x_{n-k} \pmod{2^m}, \quad j < k \quad (7)$$

Whereas the MLFG has a maximal period of $(2^k - 1)\, 2^{m-3}$, a quarter of the length of the corresponding ALFG period [10], the MLFG has empirical properties considered superior to those of the ALFG [9]. Of interest for parallel computing is the existence of a parameterization analogous to that of the ALFG for the MLFG [11].

SPRNG

The SPRNG library (http://www.ncsa.uiuc.edu/Apps/SPRNG) is designed to use parameterized pseudorandom number generators to provide random number streams to parallel processes. SPRNG is currently in its first full release (version 1.0).

The National Center for Supercomputing Applications at the University of Illinois (funded by the National Science Foundation under PACI) now supports and maintains SPRNG as part of its high-performance software activities. Most of the high-performance computing vendors have expressed considerable interest in using SPRNG as a common, parallel pseudorandom number generation library on their machines. Thus, SPRNG itself could become a lasting contribution to mathematical software for parallel Monte Carlo computations.

The SPRNG library includes the following:

- Several qualitatively distinct, well-tested scalable RNGs
- Initialization without interprocessor communication
- Reproducibility through use of 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 common RNGs to provide full-period streams of random numbers for each unique parameter value. These generators then became the core of those currently available in SPRNG:

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

Each of these 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 the streams associated with the subtree rooted at the given stream. This technique can be used for the automatic management of static and dynamic creation of streams, and it 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.

The SPRNG library was designed to be both flexible and as easy to use as possible. The Monte Carlo community is very conservative, and many groups use RNGs that have been handed down for generations (sometimes all the way back to Lehmer or Metropolis!). Thus, in addition to developing the library in collaboration with a member of the Monte Carlo community, we made the library capable of extension to include user-supplied generators. Users can add their own RNGs by rewriting two dummy routines and recompiling the library; they then have access to their own generators within the SPRNG parallel infrastructure. This is a powerful capability.

Our implementation 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 for the validation of new generators. Together, the extensibility and the testing suite aid users who want to implement their own generators in parallel and, at the same time, provide library developers with a powerful rapid prototyping tool.

Through the default generators, SPRNG is a tool for parallel pseudorandom number generation. The results obtained are reproducible, and the library provides a simple way to run on distributed-memory parallel machines, with popular languages and parallel paradigms, and also supports distribution on heterogeneous collections of machines. The developers of Condor, a

distributed-computing tool, plan to incorporate SPRNG so that Condor will be a comprehensive tool for Monte Carlo calculations on distributed heterogeneous collections of machines [7]. When a different RNG is desired, e.g., when a particular RNG is thought to give spurious results in a given application, a user can substitute a qualitatively different generator for the original by merely relinking the user program with SPRNG. Finally, new RNGs can be incorporated into SPRNG with little effort required beyond coding the generation and initialization routines and recompiling the library.

Using SPRNG: Some Examples

Extensive on-line documentation is available on the SPRNG library's Web pages. The curious reader is referred to these pages for a detailed tutorial and examples. In this article we present the bare minimum required for use of SPRNG via the "simple" interface. The definition of the `init_sprng` routine (the routine called to initialize the various random number streams for parallel use), along with a description of all the inputs required to call `init_sprng`, appears at the top of this page. The simple C code shown at right illustrates the initialization of several SPRNG RNGs in parallel with MPI, demonstrating how easy it is to replace existing RNGs with those from SPRNG.

Conclusions and Open Problems

Although care has been taken in constructing generators for the SPRNG package, we realize that there is no such thing as a PRNG that behaves flawlessly for every application. This is even more true when scalable platforms for Monte Carlo are considered. The underlying recursions used for PRNGs are simple and thus inevitably have regular structures. This deterministic regularity permits analysis of the sequences and is at the same time the Achilles heel of PRNGs.

Any large Monte Carlo calculation must be viewed with suspicion—an unfortunate interplay between the application and a PRNG can produce 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 a PRNG include the ability to use another PRNG in the same calculation. In addition, it must be possible to

```
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; typically the process number, it 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. Use of the same seed for all the streams is acceptable (and recommended).
- `param` selects the appropriate parameters, e.g., the multiplier for an LCG or the lag for a lagged-Fibonacci generator.
- `init_sprng` returns the *ID* of the stream.

```
/*
 * 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 initialization calls */

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

/* SPRNG initialization */

   init_sprng( SEED , SPRNG_DEFAULT );   /* initialize stream */
   printf("Process %d, 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 */
}
```

develop and use entirely new PRNGs. These capabilities, along with parallel and serial tests of randomness [2], are components that make the SPRNG package a useful tool for parallel Monte Carlo.

Acknowledgments

The SPRNG library 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 author's group at the University of Southern Mississippi. The author also acknowledges the support and collaboration of Steven Cuccaro, Daniel Pryor, and M.L. Robinson at the Center for Computing Sciences.

References

[1] R.P. Brent, *On the periods of generalized Fibonacci recurrences*, Math. Comput., 63 (1994), 389–401.

[2] S.A. Cuccaro, M. Mascagni, and D.V. Pryor, *Techniques for testing the quality of parallel pseudorandom number generators*, Proc. Seventh SIAM Conf. Parallel Processing for Sci. Comp., SIAM, Philadelphia, 1985, 279–284.

[3] M. Deleglise and J. Rivat, *Computing $\pi(x)$: The Meissel, Lehmer, Lagarias, Miller, Odlyzko method*, Math. Comput., 65 (1996), 235–245.

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

[5] T.G. Lewis and W.H. Payne, *Generalized feedback shift register pseudorandom number algorithms*, J. ACM, 20 (1973), 456–468.

[6] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and Their Applications*, Cambridge University Press, London, 1986.

[7] M. Litzkow, M. Livny, and M.W. Mutka, *Condor—A hunter of idle workstations*, Proc. 8th Int. Conf. Dist. Comp. Sys., June 1988, 104–111.

[8] G. Marsaglia, *Random numbers fall mainly in the planes*, Proc. Natl. Acad. Sci., USA, 62 (1968), 25–28.

[9] G. Marsaglia, *A current view of random number generators*, Comp. Sci. Stat., Proc. XVIth Sym. Interface, 1985, 3–10.

[10] G. Marsaglia and L.-H. Tsay, *Matrices and the structure of random number sequences*, Lin. Alg. Appl., 67 (1985), 147–156.

[11] M. Mascagni, *A parallel non-linear Fibonacci pseudorandom number generator*, abstract, 45th SIAM Anniversary Meeting, 1997.

[12] M. Mascagni, *Parallel linear congruential generators with prime moduli*, Par. Comp., 24 (1998), 923–936; IMA Preprint #1470, 1998.

[13] M. Mascagni, S.A. Cuccaro, D.V. Pryor, and M.L. Robinson, *A fast, high-quality, and reproducible lagged-Fibonacci pseudorandom number generator*, J. Comp. Phys., 15 (1995), 211–219.

[14] 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 (1995), 263–277.

[15] J.L. Massey, *Shift-register syn-thesis and BCH decoding*, IEEE Trans. Inform. Theory, IT–15 (1969), 122–127.

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

[17] O.E. Percus and M.H. Kalos, *Random number generators for MIMD parallel processors*, J. Par. Dist. Comp., 6 (1989), 477–497.

Michael Mascagni (mascagni@cs.fsu.edu) is an associate professor in the Department of Computer Science at Florida State University in Tallahassee.