Parameterizing Parallel Multiplicative Lagged-Fibonacci Generators

Michael Mascagni and Ashok Srinivasan
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4530 USA
E-mail: {mascagni,asriniva}@cs.fsu.edu

Abstract

Monte Carlo computations are commonly considered to be naturally parallel. However, one needs to exercise care in parallelizing the underlying **p**seudo**r**andom **n**umber **g**enerator (PRNG) to avoid correlations within, and between, random number streams. PRNGs are normally parallelized using one of the following two paradigms: (i) cycle division and (ii) parameterization. Most of the popular PRNGs based on linear recurrences, such as **l**inear **c**ongruential **g**enerators (LCGs), **g**eneralized **f**eedback **s**hift-**r**egister **g**enerators (GFSRs), and **a**dditive **l**agged-**F**ibonacci **g**enerators (ALFGs) have been parallelized using both these paradigms. While the (non-linear) **m**ultiplicative **l**agged-**F**ibonacci **g**enerators (MLFG) is considered superior to the ALFG in quality, it had so far been parallelized only via the cycle division paradigm, which has its limitations. In this paper, we describe parallelization of the MLFG using parameterization, and discuss its implementation in the **S**calable **P**arallel **R**andom **N**umber **G**enerators (`SPRNG`) [20] parallel pseudorandom number generation software. We also present empirical results demonstrating the quality, and quantitatively compare the parallel ALFG's and MLFG's quality.

1 Introduction

Pseudo**r**andom **n**umber **g**enerators (PRNGs) are required in variety of applications, with **M**onte **C**arlo (MC) computations being perhaps the most commonly known. Correlations in the pseudorandom number sequence can lead to errors in the results of MC computations, as discussed in several studies [10, 11, 24, 25]. In parallel Monte Carlo applications, each process generates a distinct stream of pseudo random numbers. And so we need to be concerned not just with the *intra-stream correlations*, as mentioned above, but also with *inter-stream* correlations between the streams being generated on different processors. With the increasing popularity of MC in high-performance and distributed applications, due in large measure to its ease of parallelization, the quality of the underlying **p**arallel **p**seudo**r**andom **n**umber **g**enerator (PPRNG) has become even more critical. Because of this, the testing of PPRNGs has been well studied, and has been discussed in [3, 4, 5, 8, 9, 23, 26, 27]. Moreover, defects in popular PPRNGs, leading to errors in practical applications, have also been demonstrated. This underlines the delicacy in the task of selecting, analyzing, parallelizing, testing, and finally deploying high-quality PPRNGs.

In view of its importance, much work has been done on parallelizing popular PRNGs, and studying the properties of the proposed parallelizations. The techniques used in parallelizing PRNGs can be broadly classified as (i) *cycle division* and (ii) *parameterization*. In all the methods of cycle division, a single PRNG sequence, which can be mentally pictured as a periodic circle of random numbers, is partitioned amongst the different processors. This partitioning is generally performed in one of the following ways. *Naïve* users randomly select seeds on each processor and hope that the seeds will take them to widely separated portions of the sequence, so that there will be no overlap between the sub-sequences used by different processors. In the *blocking* scheme, the user determines a block size, B. If the PRNG sequence is given by $x_0, x_1, \ldots$, then the PPRNG sequence on processor i is given by $x_{iB}, x_{iB+1}, \ldots$. The danger here is that if the user happens to consume more random numbers than expected, then the streams on different processors could overlap. More importantly, PRNGs often have long-range correlations [7, 6], which turn into short-range inter-stream correlations in the PPRNG constructed in this manner. (Short-range correlations are considered more dangerous for typical applications than are long-range ones.) In the *leap frog* scheme, processor i gets the sequence $x_i, x_{i+P}, x_{i+2P} \ldots$, where P is the total number of processors. Long range correlations in the PRNG become short-range intra-stream correlations in this PPRNG, which can be worse than inter-stream correlations. A final approach using cycle division is a method known as recursive-halving leap-ahead. In this case, the cycle is dynamically partitioned as new parallel streams are needed. While this minimizes the chance of substream overlap, it does not lessen the generic problem that pieces of the same full-period PRNG can be highly correlated to other pieces. Apart from these defects, the method of cycle-division results in a non-scalable PPRNG.[1] Since we are taking all the random numbers used in a single computation from a single PRNG, the total numbers available in a computation is the period of the PRNG. To solve a bigger problem could require more numbers forcing the use of a generator with a longer period. Since the cost of generating a random number from a PRNG with period P is related to P and often as big as $O(\log_2 P))$, solving a bigger problem will result in an increase in the cost of generating each random number. This will occur even if the number of random numbers used on a single processor of the computation stays fixed as problem size grows. This lack of scalability is the nature of cycle-division methods; however, they are not problems when using the parameterization technique described next. The interested reader will find a more detailed description of defects to be found in cycle-division based schemes demonstrated in [20].

Parallelization schemes based on parameterization, on the other hand, provide independent, full-period sequences on each processor. In this method, we have access to a large set of pseudorandom sequences. Parameterization of PPRNGs is based on the fact that given a sequence number i, there is an easy way of generating the ith sequence. This type of parameterization can be done in two ways. *Seed parameterization* is used with certain generators where the set of possible initial states of a PRNG naturally divides itself into a number of smaller cycles. We can number these cycles from 0 to $N-1$, where N is the total number of cycles available. We then give each processor a seed from a different cycle. In *iteration function*[2] *parameterization*, a different iteration function is used for each stream. In order to achieve this, we need a way of parameterizing the iteration function so that given i, the ith iteration function can be produced easily.

Most of the popular PRNGs based on linear recurrences, such as the LCG, GFSR generator, and the ALFG have been parallelized via both paradigms (cycle division and parameterization). Linear recurrences have the disadvantage that points from these generators fall along hyperplanes[15]. Therefore one would prefer to use a generator based on a nonlinear recurrence. The MLFG is a nonlinear generator that has empirically been shown to have good statistical properties. In particular, it has much better statistical properties than its linear cousin, the ALFG [16]. Makino [13] has presented a parallelization of the MLFG using cycle-division methods. In this paper, we describe a parallelization of this generator through parame-

[1] Each PRNG has a finite number of possible states, and hence the "random" sequence will start repeating after a certain "period," causing a cycle [26].

[2] Given the current state of the generator, the iteration function defines the next state [26].

terization.

We begin our paper with a description of the ALFG, the MLFG, and their properties in § 2. We next describe Makino's [13] parallelization of the MLFG in § 3. Our parallelization of the MLFG is related to our ALFG parallelization based on parameterization. We therefore first summarize the procedure for the parameterized parallelization of the ALFG in § 4. We then present our new parallelization of the MLFG in § 5. In § 6 we presents results of empirical tests on the MLFG. Here we compare the MLFG specifically to the ALFG in terms of statistical quality. Finally, in § 7 we summarize our results, provide concluding comments, and indicate future directions for this research.

2 Lagged-Fibonacci generators

2.1 Additive lagged-Fibonacci generators

The recurrence relation for the ALFG is given by the following equation:

$$x_n = x_{n-k} + x_{n-l} \ (mod\ 2^b), \tag{1}$$

where l and k are called the lags of the generator, and we use the convention that $l > k$. The random number stream generated is defined by the seed, consisting of the l initial values, $x_0, x_1, ..., x_{l-1}$. For suitably chosen values of the lags and seed [17, 13], this PRNG attains its maximal period of $P = 2^{b-1}(2^l - 1)$.

2.2 Multiplicative lagged-Fibonacci generators

The recurrence relation for the MLFG is given by the following equation, which is closely related to that for the ALFG:

$$x_n = x_{n-k} \times x_{n-l} \ (mod\ 2^b). \tag{2}$$

Its maximal period is $P = 2^{b-3}(2^l - 1)$, and in § [13] we explain under what conditions this period can actually be achieved.

3 Parallelization of MLFGs through cycle-division

Makino [13] parallelized the MLFG based on the following observation of Marsaglia [16]. Since the product of an even number and another number always produces an even number, if an MLFG has a seed with as little as a single even number in it, the MLFG sequence eventually becomes all even. However, if we demand that all integers in the MLFG seed start out odd, then they remain odd, and more importantly, there is no initial transient in the sequence of its approach to being uniformly odd as there is in the approach to uniform evenness. Therefore, it is customary to seed MLFGs with only odd integers. One needs to be aware of this fact, as the use of the least-significant bit of an MLFG, which is always one in this case, should be done prudently with full knowledge of this limitation. Given that we use only odds we note that any odd integer modulo 2^b, x, can be uniquely represented as $x = (-1)^y 3^z (mod\ 2^b)$, where $y \in \{0,1\}$ and $z \in \{0,1,...,2^{b-2}-1\}$. Substituting these relationships into equation (2), we get the following equation for the MLFG:

$$x_n = (-1)^{y_n} 3^{z_n} \ (mod\ 2^b), \tag{3}$$

where y_n is given by the recurrence

$$y_n = y_{n-k} + y_{n-l} \ (mod\ 2), \tag{4}$$

and z by the recurrence

$$z_n = z_{n-k} + z_{n-l} \ (mod\ 2^{b-2}). \tag{5}$$

Equations (4) and (5) are recurrences for ALFGs with periods $2^l - 1$ and $2^{b-3}(2^l - 1)$ respectively. If l initial values for the y_n and z_n sequence are given such that the z_n sequence has its maximal period, then the corresponding l initial values for the sequence x_n can be obtained, which therefore defines a MLFG sequence with period $2^{b-3}(2^l-1)$. Makino then parallelizes via cycle division and blocking by choosing the seeds so that ith random number stream will produce the sequence x_{Bi}, x_{Bi+1}, $\ldots$, where B is the block-size. If B is larger than the number of random numbers consumed from any stream, then there will be no overlap between the sequences produced on the different processors.

Since B needs to be large to prevent overlap, it is not feasible for processor i to generate the entire subsequence $x_0, \ldots, x_{Bi}, \ldots, x_{Bi+l-1}$ to start its sequence; so a "jump-ahead" facility needs to be provided that enables fast computation of $x_{Bi}, \ldots x_{Bi+l-1}$, given $x_0, \ldots, x_{l-1}$. Makino's scheme consists of, equivalently, choosing seed $y_0, \ldots, y_{l-1}$ and maximal period seed $z_0, \ldots, z_{l-1}$, and then computing $y_{Bi}, \ldots y_{Bi+l-1}$ and $z_{Bi}, \ldots z_{Bi+l-1}$ efficiently in $O(l^2 \log Bi)$ time using Miller and Brown's [21] algorithm. Once the seeds for the y_n and z_n sequences of each stream are computed, the seeds for the x_n sequence are easily obtained from equation (3). Subsequent random numbers are obtained using the recursion (2).

We remind the reader that the above scheme is a cycle-division scheme, based on blocking, and therefore has the potential pitfalls mentioned in § 1. In addition, it is interesting to remark on the relationship between ALFG and MLFG random numbers that this approach uses. The identity $x = (-1)^y 3^z (mod\ 2^b)$ for odd integers modulo a power-of-two allows us to consider ALFG random numbers as the exponents in a discrete logarithmic representation of the corresponding MLFG numbers. In this case the "base" for our discrete logarithms modulo 2^b is 3. It is not surprising that in the case of modular recurrences that the multiplicative and additive versions are related via exponentiation and the discrete logarithm.

4 Parallelization of ALFGs through parameterization

The ALFG can be parallelized in the cycle-division paradigm using Aluru *et al.*'s [1] algorithm for leap-frogging, or Makino's scheme (based on Miller and Brown's algorithm [21]) [13] for blocking. However, we have mentioned disadvantages of this paradigm earlier. Alternatively, the ALFG can be parallelized through parameterization, as summarized below.

The parallelization of the ALFG through seed parameterization is based on the observation that there are $2^{(b-1)(l-1)}$ distinct maximal period cycles available. We therefore try to place a different cycle (defined by its seed) on each processor. Of course, placing distinct seeds on each processor does not guarantee distinct cycles, since processors could have different seeds that come from the same cycle.

It is possible to systematically choose seeds in such a manner that each processor gets a seed from a different cycle. Mascagni *et al.* [18] observed that the state space of the ALFG is toroidal, with equation (1) providing the algorithm for movement in one of the torus dimension. Finding the algorithm for movement in the other dimension is the basis of their parameterization scheme. Since equation (1) 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.

An elegant algorithm for movement in this second dimension is based on a simple enumeration as follows. One can prove that the initial seed,

$\{x_0, x_1, \ldots, x_{l-1}\}$, can be bit-wise initialized using the following template:

$$
\begin{array}{cccc|c|c}
\text{m.s.b.} & & & & \text{l.s.b.} & \\
b_{b-1} & b_{b-2} & \ldots & b_1 & b_0 & \\
\hline
\blacksquare & \blacksquare & \ldots & 0 & 0 & x_{l-1} \\
0 & \blacksquare & \ldots & \blacksquare & 0 & x_{l-2} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \\
\blacksquare & 0 & \ldots & \blacksquare & 0 & x_1 \\
\blacksquare & \blacksquare & \ldots & \blacksquare & 1 & x_0
\end{array}
\tag{6}
$$

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. [18]. Note that here the least-significant bits, b_0 are specified to be a fixed, non-zero, pattern. If one allows an $O(l^2)$ precomputation to find a particular least-significant-bit pattern then the template is particularly simple:

$$
\begin{array}{cccc|c|c}
\text{m.s.b.} & & & & \text{l.s.b.} & \\
b_{b-1} & b_{b-2} & \ldots & b_1 & b_0 & \\
\hline
\blacksquare & \blacksquare & \ldots & \blacksquare & b_{0k-1} & x_{l-1} \\
\blacksquare & \blacksquare & \ldots & \blacksquare & b_{0k-2} & x_{l-2} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \\
\blacksquare & \blacksquare & \ldots & \blacksquare & b_{01} & x_1 \\
\hline
0 & 0 & \ldots & 0 & b_{00} & x_0
\end{array}
\tag{7}
$$

Furthermore, one can provide evidence for the absence of correlation between streams, based on theorems proven on the behavior of the exponential sum correlations test [20]. 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.* [19]. However, it is easy to avoid all but the most uncorrelated pairs in a computation [22]. In this case there is extensive empirical evidence that the full-period exponential sum correlation between streams is $O\left(\sqrt{P = (2^l - 1)2^{b-1}}\right)$, 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 [19], is an important open problem in the theory of ALFGs.

5 Parallelization of the MLFG through parameterization

Our parallelization of the MLFG is based on the application of equations (3), (4) and (5) to the ALFG parameterization scheme described in § 4. We initialize streams through the following process. (i) We select a seed for the y_n sequence by choosing initial values for the l bits in its state. (ii) We choose a seed for the $b-2$ bit ALFG sequence, z_n, using the ALFG algorithm [18] outlined earlier, to ensure that each seed is from a distinct cycle. (iii) We then use Equation (3) to get the corresponding seeds for the MLFG sequence, $x-N$. Subsequent elements of the sequence can be obtained by directly applying the recurrence (2).

The number of distinct full-period cycles for z_n is $2^{(b-3)(l-1)}$, and for each of these, we can start with 2^l possible values of the y_n yielding $2^{(b-2)(l-1)+1}$ distinct full-period cycles for the MLFG[3].

The parallel MLFG implemented in the SPRNG library uses $b = 64$ to obtain a period of $P = 2^{61}(2^l - 1)$. However, the number of distinct streams is not $2^{62(l-1)+1}$, but is only $2^{61(l-1)}$, for the following reason. Let us consider a set of MLFG sequences initialized with identical z_n's, considering only the possibility of the y_n components of the seed varying. The numbers produced by each sequence will by identical, except for a difference in sign, modulo 2^b. In the corresponding floating point random numbers, if the ith element of one of these related sequences is x_i, then the ith element of all the other related sequences will be either x_i or $1-x_i$. Whether this leads to dangerous correlations is an open question. To be on the safe side, SPRNG keeps the initial y_n sequence fixed for all theses sequences, and therefore ensures that the initial z_n seeds are all distinct. In SPRNG , the lag, l, varies from the default value of 17 to 1279. The period, for $l = 17$, is approximately 3×10^{23}, and the number of available streams is approximately 6×10^{293}.

6 Empirical test results

While the quality of a PRNG sequence is extremely important, significant aspects of a generator's quality are hard to prove mathematically. Therefore, the ultimate test of a PRNG's quality is empirical. Empirical tests of randomness fall broadly into two categories (i) statistical tests and (ii) application-based tests.

[3]Note that when we seed z_n based on the algorithm in [18], each distinct seed for the y_n sequence will lead to a distinct MLFG cycle, even if the z_n seeds are identical.

Statistical tests compare some statistic obtained using a PRNG with what would be obtained with truly random numbers. The tests assume that the truly random number are independent and identically distributed numbers in the unit interval. If the two results are very different, then the PRNG is considered defective. Certain statistical tests [12, 14] have become de-facto standards for sequential PRNGs. Sequential and parallel versions of most of these standard tests are included in the `SPRNG` [26] test suite, which we used to verify the quality of the MLFG.

Application-based tests rely on the observation that certain applications interact with PRNGs in unpredictable and unsatisfactory ways. Thus, statistical tests and theoretical results are not adequate to demonstrate the quality of a PRNG. Application-based tests use random numbers in a manner similar to the way they are used in common MC applications, except that the exact solutions to the test applications are known. However, unlike with statistical tests, the results of an application-based test are not known to be drawn from a particular, known, statistical distribution. Thus, one must use experience and familiarity with the application to decide whether the results obtained "pass" or "fail" the test. We present results of the `SPRNG` application-based tests based on computing critical point information from the Ising model from statistical physics. This test is known to be exquisitely sensitive to correlations, and is considered very revealing. Further details on these tests can be found in [26], and a summary in Appendix A.

A PPRNG needs to be a good sequential PRNG as well, and therefore we first present the results of sequential PRNG tests on the MLFG, followed by results of the PPRNG tests, in subsection 6.1. Subsequently, in subsection 6.2, we compare the ALFG and MLFG in terms of their statistical quality, and give a guide to choosing parameters for these generators such that their quality is roughly the same.

6.1 Tests on the `SPRNG` MLFG

Sequential tests: The DIEHARD [14] test suite runs the following tests to verify the randomness of approximately three million random numbers: the birthday spacings test, overlapping 5-permutation test, binary rank test (for 6x8, 31x31 and 32x32 matrices), bitstream test, overlapping pairs sparse occupancy (OPSO), overlapping quadruples sparse occupancy (OQSO), DNA, count-the-1's test (for a stream of bytes and for specific bytes), parking lot test, minimum distance test, 3dspheres test, squeeze test, overlapping sums test, runs test, and the craps test. All the tests in the DIEHARD suite are passed for our `SPRNG` MLFGs, even with a small lag of 17.

The SPRNG MLFG has several variants, corresponding to different possible lags. We tested the first 1024 streams of each of these variants with the sequential statistical tests from the SPRNG test suite, and with the random walk test [28]. The parameters were chosen so as to test around 10^{11} random numbers, except for the collisions test, which used 10^{12} random numbers. We chose to make the latter test was larger because the collisions test is particularly effective in detecting defects. The random walk test used just around 10^8 random numbers, since we have implemented a rather simple version of it. The details of the parameters are summarized in Table 1, indicating that all the tests were passed by the SPRNG MLFG. In this table we publish the "K-S Percentile" obtained from the statistical test. This value is the percentile of where the empirical distribution lies within the expected distribution of the Kolmogorov-Smirnov (K-S) statistic that is commonly used to measure the difference between two cumulative probability distributions. A value of 50 for this means that the empirical results from the MLFG were exactly at the median of the distribution expected from truly random numbers. Thus a test fails when this value is too close to the extreme percentiles of 0 and 100.

The application-based Ising model tests were performed with the Metropolis and Wolff algorithms on a 16×16 lattice. The block size was taken to be 1000, and the results of 1000000 blocks were considered, after discarding the results of the first 100 blocks. We do this so as to discard an initial transient in a simulation that endeavors to sample a statistic from an equilibrium distribution. This test consumes a block of 10^{11} random numbers. The generator passed both these tests as shown in Figures 1 and 2. These figures show the plot for the computed specific heat with the Metropolis algorithm, and with the energy obtained for the Wolff algorithm. We actually tested both energy and specific heat for both the algorithms.

Parallel tests: For those tests from the SPRNG test suite that needed interleaved streams, we created four streams, with each stream being the result of interleaving 256 streams. Each of these was subjected to the standard tests. The details of the parameters and results are given in Table 2, indicating that all the tests were passed. The number of random numbers consumed in each test was around 10^{11}, unless mentioned otherwise. As with the sequential tests, we used approximately 10^{12} random numbers with the collisions test, since it is very effective. The equidistribution test is redundant in the parallel version, since interleaving streams does not make any difference in terms of the equidistribution over the sequential test.

The Ising model tests were performed with the Metropolis and Wolff

Test	Parameters	K-S Percentile
Collisions	$n = 100000$, $\log md = 10$, $\log d = 3$	71.8
Collision	$n = 200000$, $\log md = 4$, $\log d = 5$	90.8
Coupon	$n = 5000000$, $t = 30$, $d = 10$	19.6
Equidist	$d = 10000$, $n = 100000000$	5.9
Gap	$t = 200$, $a = 0.5$, $b = 0.51$, $n = 1000000$	87.1
Maxt	$n = 50000$, $t = 16$	59.7
Permutations	$m = 5$, $n = 20000000$	78.8
Poker	$n = 10000000$, $k = 10$, $d = 10$	88.4
Random walk	$walk\ length = 1024$	96.4
Runs	$t = 10$, $n = 50000000$	47.5
Serial	$d = 100$, $n = 50000000$	45.9

Table 1: *Sequential PRNG tests on MLFG. The common test parameters (see Appendix A for explanation) were:* $nstreams = 1024$, $ncombine = 1$, $seed = 0$, $nblocks = 1$, $skip = 0$, *with the following exceptions: (i) The collisions tests used* $nblocks = 1000$. *This test was done twice, with* $seed = 9999$ *in the first instance above and* $seed = 0$ *in the second. (ii) the* $Maxt$ *and random walk tests used* $nblocks = 100$ *(iii) The* $Maxt$ *test used* $seed = 9999$. *The K-S test percentile given above is for the default* `SPRNG` *MLFG with lags* $17, 5$.

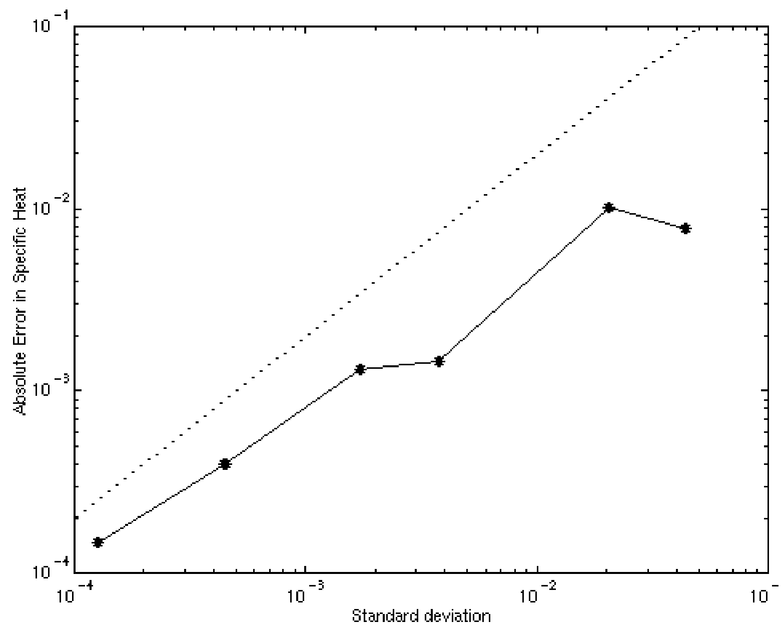

Figure 1: Plot of the actual error versus the internally estimated standard deviation of the specific heat for Ising model simulations with the Metropolis algorithm on a 16×16 lattice with a sequential MLFG with lags $(17, 5)$. We expect around 95% of the points to be below the dotted line (representing an error of two standard deviations) with a good generator.

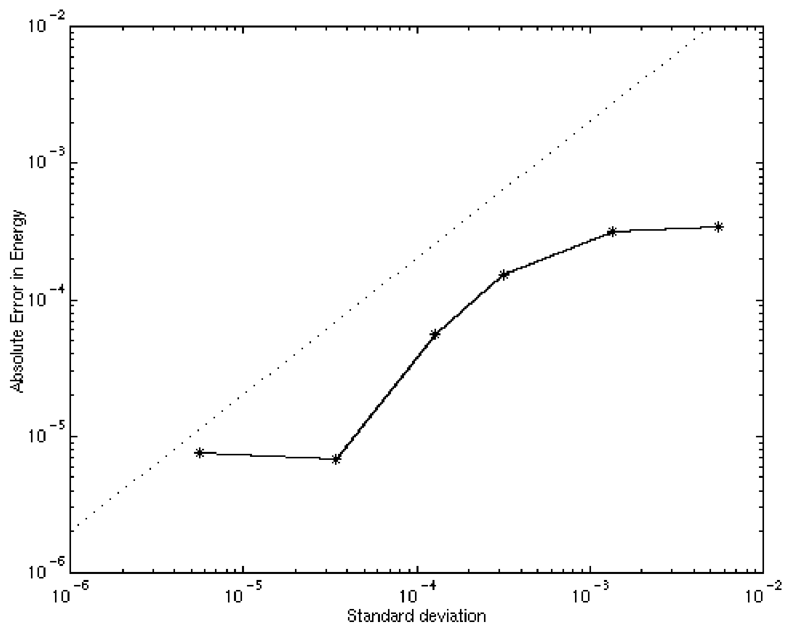

Figure 2: Plot of the actual error versus the internally estimated standard deviation of the energy for Ising model simulations with the Wolff algorithm on a 16×16 lattice with a sequential MLFG with lags $(17, 5)$. We expect around 95% of the points to be below the dotted line (representing an error of two standard deviations) with a good generator.

Test	Parameters	K-S Percentile
Blocking	$n = 1000000$, $r = 128$	10.2
Collision	$n = 200000$, $\log md = 10$, $\log d = 3$	60.6
Collision	$n = 200000$, $\log md = 4$, $\log d = 5$	46.8
Coupon	$n = 5000000$, $t = 30$, $d = 10$	12.9
Gap	$t = 200$, $a = 0.5$, $b = 0.51$, $n = 1000000$	78.0
Permutations	$m = 5$, $n = 20000000$	7.2
Poker	$n = 10000000$, $k = 10$, $d = 10$	9.7
Random walk	$walk\ length = 1024$	6.8
Runs	$t = 10$, $n = 50000000$	33.2
Serial	$d = 100$, $n = 50000000$	31.2

Table 2: *Parallel PRNG tests on the MLFG. The common test parameters were: $nstreams = 4$, $ncombine = 256$, $seed = 0$, $nblocks = 250$, $skip = 0$, with the following exceptions: (i) The blocking test does not do interleaving, and used the 1024 streams directly. (ii) The collisions test was performed twice. In the first one the parameters $nstreams = 32$, $ncombine = 32$, and $nblock = 160$ were used, while for the second $nstreams = 16$, $ncombine = 64$, and $nblocks = 50000$ were used. (iii) the random walk test used $nblocks = 25000$ for around 10^8 random numbers. The K-S test percentile given above is for the default MLFG lags of $17, 5$.*

algorithms on a 16×16 lattice. The block size was taken to be 1000, and the results of 1000000 blocks were considered, after discarding the results of the first 100 blocks. This tests over 10^{11} random number. Figure 3 shows the plot for the specific heat with the Metropolis algorithm.

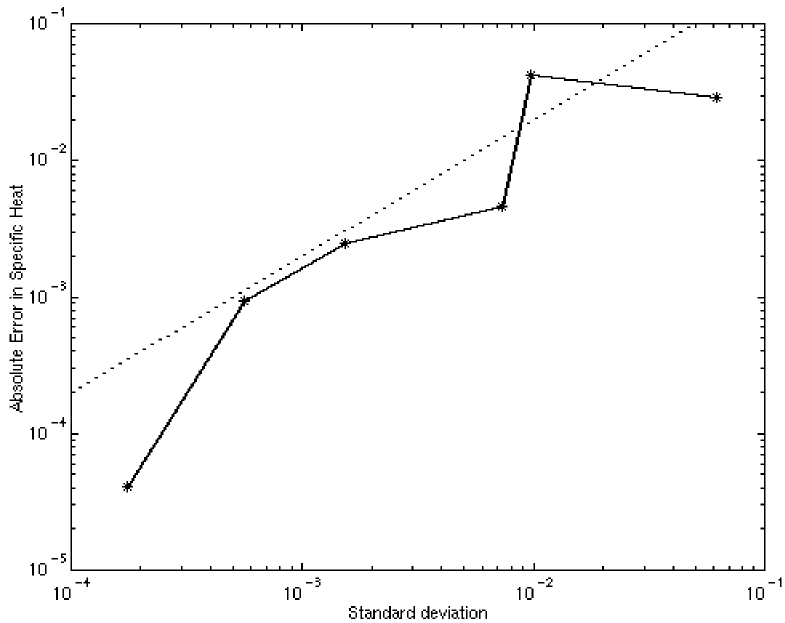

Figure 3: Plot of the actual error versus the internally estimated standard deviation of the specific heat for Ising model simulations with the Metropolis algorithm on a 16×16 lattice with a sequential MLFG with lags $(17, 5)$. We expect around 95% of the points to be below the dotted line (representing an error of two standard deviations) with a good generator.

6.2 Comparison of the ALFG to the MLFG

We now compare the relative quality of the MLFG and ALFG with different lags. The lags considered where those used in the `SPRNG` package, namely: $(17, 5)$, $(31, 6)$, $(55, 24)$, $(63, 31)$, $(127, 97)$, $(521, 353)$, $(521, 168)$, $(607, 334)$, $(607, 273)$, $(1279, 418)$, $(1279, 861)$.

We first performed tests with the birthday spacings test from DIEHARD, since it is well known that ALFGs are especially susceptible to it. Since the number of random numbers tested by DIEHARD is fixed, we instead chose to vary the number of bits (corresponding to varying b in Equations (1) and (2) for the ALFG and MLFG respectively), and observed the number of bits at which each generator failed. In the tests of the sequential version of the generators, the following ALFG lags failed up to 64 bits (the largest one tested): $(17,5)$, $(31,6)$, $(55,24)$, $(63,31)$, $(127,97)$. The ALFG lags of $(521,353)$, $(521,168)$, $(607,334)$, $(607,273)$, $(1279,418)$, $(1279,861)$ failed when the number of bits was less than 33. On the other hand, all the MLFG lags failed when the number of bits was less than around 33 or 34 bits. Thus in the sequential tests performed, MLFG lags of 17 and larger gave roughly similar performance to ALFG lags of 521 and larger, while ALFG lags of 127 and smaller performed uniformly poorly.

We also performed tests with the *gap* test with parameters $t = 50$, $a = 0.5$, $b = 0.6$, and $n = 100000$. We kept the number of bits fixed at 32, but varied the number of random numbers tested, and observed the point at which the tests failed. However, the conclusions were weaker than with the DIEHARD test above. The ALFG failed for lag $(17,5)$ with around 10^7 random numbers, while with lag $(31,6)$ it failed for around 10^8 random numbers. However, other ALFG lags, and all the MLFG lags, passed with up to 10^{11} random numbers.

We next performed parallel tests, where we interleaved 1024 streams to create a new random number stream, which we tested with the birthday spacings test from DIEHARD. Now, all the ALFG lags failed up to 64 bits (the largest tested), while the MLFG lags failed for less than around 33 or 34 bits. Thus it appears that the ALFG has strong inter-stream correlations, which is not apparent in the MLFG.

We finally performed the parallel gap test with 1024 streams interleaved, with the other test parameters remaining $t = 50$, $a = 0.5$, $b = 0.6$, and $n = 100000$, and with the number of bits being 32, as above. The ALFG failed for all lags with around 10^6 random numbers, while all the MLFG lags passed with up to 10^{11} random numbers. In fact, the ALFG failed for all lags, even if we interleaved just 128 streams!

Thus we conclude, based on these tests that, in sequential computations, MLFG lags of 17 and above appear roughly equivalent to ALFG lags of 521 and above, while in parallel applications, the ALFG is not suitable, while the MLFG performs well. (We note that the ALFG in the `SPRNG` software is a modified ALFG, and it passes all the parallel tests we performed.)

7 Conclusions

Most parallelizations of PRNGs, and all prior parallelizations based on parameterization, were developed for PRNGs based on linear recurrences. The MLFG, which follows a nonlinear recurrence, was parallelized by Makino [13] based on cycle-splitting, whose limitation we have mentioned earlier. We have developed a parallelization scheme for the MLFG using parameterization, and conducted extensive empirical tests that verify its quality.

As shown earlier, the MLFG is closely related to the ALFG. However, ALFGs have certain correlations, within and between streams, that cause them to fails some standard tests, such as the Birthday Spacings test from DIEHARD, the gap test [12], and the application-based test designed from the Wolff algorithm for the Ising model [26]. The ALFG implemented in the `SPRNG` package actually combines two different random number streams to produce a new one, in order to avoid this problem. In contrast, the MLFG passes these tests without requiring such modifications, and appears safe for use even with small lags.

With the addition of the MLFG to the `SPRNG` library, `SPRNG` now contains a very high quality nonlinear generator. Being nonlinear, the MLFG is qualitatively different from all the other currently available `SPRNG` generators. This is important as the designers of `SPRNG` have always advocated the use of qualitatively different generators as the basis for empirical validation of large, complicated parallel MC computations. In addition, the empirical test evidence shows that the MLFG with very small lags performs as well as the ALFG with quite large lags. Since the cost of initialization and generation of MLFG random numbers is the same as for the ALFG, the MLFG offers the opportunity to replace the ALFG in applications where requirements to minimize the memory needed for the PRNG seed are important.

Acknowledgment

This parallelization of the MLFG was first implemented and tested as a part of the `SPRNG` software, which 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.* We also wish to acknowledge the computational resources provided by NCSA, University of Illinois at Urbana-Champaign. `SPRNG` is now funded through a Department of Energy Accelerated Strategic Computing Initiative (ASCI) Level 3 contract sponsored by Lawrence

Livermore, Los Alamos, and Sandia National Laboratories.

References

[1] S. Aluru, G. M. Prabhu, and J. Gustafson. A random number generator for parallel computers. *Parallel Computing*, 18:839–847, 1992.

[2] P.D. Beale. Exact distribution of energies in the two-dimensional Ising model. *Phys. Rev. Lett.*, 76:78, 1996.

[3] P. Coddington. Analysis of random number generators using Monte Carlo simulation. *Int. J. of Mod. Phys. C*, 5(3):547–560, 1994.

[4] P. Coddington. Tests of random number generators using Ising model simulations. *Int. J. of Mod. Phys. "C"*, 7(3):295–303, 1996.

[5] 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*, pages 279–284, Philadelphia, Pennsylvania, 1995. SIAM.

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

[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. Controlling correlations in parallel Monte Carlo. *Parallel Computing*, 21:73–84, 1995.

[9] M. J. Durst. Testing parallel random number generators. In *Computing Science and Statistics: Proceedings of the XXth Symposium on the Interface*, pages 228–231, 1988.

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

[11] P. Grassberger. On correlations in 'good' random number generators. *Phys. Lett. A*, 181(1):43–46, 1993.

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

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

[14] G. Marsaglia. Diehard software package. ftp://stat.fsu.edu/pub/diehard.

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

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

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

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

[19] 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.

[20] M. Mascagni and A. Srinivasan. `SPRNG`: a scalable library for pseudorandom number generation, 2000.

[21] J. C. P. Miller and D. J. S. Brown. An algorithm for evaluation of remote terms in a linear recurrence sequence. *Comp. J.*, 9:188–190, 1966.

[22] 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*, pages 311–319. IEEE, 1994.

[23] J. Saarinen, K. Kankaala, T. Ala-Nissila, and I. Vattulainen. On random numbers - test methods and results. Preprint series in theoretical physics HU-TFT-93-42, Research Institute for Theoretical Physics, University of Helsinki, 1993.

[24] F. Schmid and N. B. Wilding. Errors in Monte Carlo simulations using shift register random number generators. *Int. J. Mod. Phys. C*, 6(6):781–787, 1995.

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

[26] A. Srinivasan, D.M. Ceperley, and M. Mascagni. Random number generators for parallel applications. In D. Ferguson, J. I. Siepmann, and D. G. Truhlar, editors, *Monte Carlo Methods in Chemical Physics*, volume 105 of *Advances in Chemical Physics*, pages 13–36. John Wiley and Sons, New York, 1999.

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

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

A Description of random number tests

We now summarize the tests performed on the (P)PRNGs. More detail on these can be found in [26].

A.1 Statistical tests

We can modify sequential random number tests to test PPRNGs by interleaving different parallel streams to produce a single new random number stream. This new stream is then subjected to the standard sequential tests. For example if stream i is given by $x_{i0}, x_{i1}, \cdots$, $0 \leq i < N$, then the new stream is given by $x_{00}, x_{10}, \cdots x_{N-1,0}, x_{01}, x_{11}, \cdots$. If each of the individual streams is random, and the streams are independent of each other, then the newly formed stream too should be random.

The arguments passed in calls to the `SPRNG` tests-suite tests all have the following format:

```
nstreams ncombine seed param nblocks skip test_args
```

The argument *ncombine* indicates the number of streams we interleave to form a new stream. We form *nstreams* such new streams and test *nblocks* blocks of random numbers from each new stream. The argument *seed* is an encoded seed to the random number generator, and *param* is the random number generator parameter [20] (for example, the lags in the ALFG and MLFG) for the generator. The argument *skip* 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, *test_args*, are specific to that test. Note that we can perform tests on individual streams (tests for intra-stream correlations) by setting *ncombine* to 1.

We next briefly describe each test and provide a list of its test specific arguments. The details concerning these tests are presented in Knuth [12], unless we mention otherwise. The (P)PRNGs were also tested with the DIEHARD test suite, including the parallel tests using interleaving.

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 then 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 originally proposed by Knuth.

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.

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.

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 single category in our analysis.

5. *Maximum-of-t test (Maxt): 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 x^t.

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 rankings 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.

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.

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.

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.

The following test is inherently parallel, in contrast to the above scheme which is really parallelization of sequential tests.

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 n such sums and check for normality. (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.)

A.2 Application-based tests

1. *Ising model – Metropolis and Wolff algorithms:* For statistical mechanical applications, the two-dimensional Ising model (a simple lattice spin model) is often used, since the exact answer for quantities such as energy and specific heat are known [2]. 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 those of Metropolis and Wolff, 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 for testing random number generators, and has often detected subtle defects in generators [3, 4, 10, 25, 28].

We can test parallel generators on the Ising model application by assigning distinct random number sequences to different subsets of lattice sites [4]. This is essentially the domain decomposition method of MC parallelization, and is more effective in determining inter-stream correlations than the replication method. In our tests of PPRNGs, we assign a distinct stream to each lattice site, thus testing the independence of a large number of streams simultaneously.

2. *Random walk test:* The random walk test implemented is a simple one, based on the algorithm in reference [28]. 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. The user needs to specify the length of a walk, apart from the common arguments as for the other statistical tests. PPRNGs are tested here by interleaving streams.