Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Parallel Computing 29 (2003) 69–94

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Testing parallel random number generators

Ashok Srinivasan [a], Michael Mascagni [b,*], David Ceperley [c]

[a] *Department of Computer Science, Florida State University, Tallahassee, FL 32308-4530, USA*
[b] *Department of Computer Science, Florida State University, Tallahassee, FL 32308-4530, USA*
[c] *National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA*

Received 20 January 2001; received in revised form 15 March 2002; accepted 29 June 2002

Abstract

Monte Carlo computations are considered easy to parallelize. However, the results can be adversely affected by defects in the parallel pseudorandom number generator used. A parallel pseudorandom number generator must be tested for two types of correlations—(i) intra-stream correlation, as for any sequential generator, and (ii) inter-stream correlation for correlations between random number streams on different processes. Since bounds on these correlations are difficult to prove mathematically, large and thorough empirical tests are necessary. Many of the popular pseudorandom number generators in use today were tested when computational power was much lower, and hence they were evaluated with much smaller test sizes.

This paper describes several tests of pseudorandom number generators, both statistical and application-based. We show defects in several popular generators. We describe the implementation of these tests in the SPRNG [ACM Trans. Math. Software 26 (2000) 436; SPRNG—scalable parallel random number generators. SPRNG 1.0—http://www.ncsa.uiuc.edu/Apps/SPRNG; SPRNG 2.0—http://sprng.cs.fsu.edu] test suite and also present results for the tests conducted on the SPRNG generators. These generators have passed some of the largest empirical random number tests.

Keywords: Parallel random number generators; Random number tests; Parallel algorithms; Random number software

* Corresponding author.
E-mail addresses: asriniva@cs.fsu.edu (A. Srinivasan), mascagni@cs.fsu.edu (M. Mascagni), ceperley@ncsa.uiuc.edu (D. Ceperley).

0167-8191/02/$ - see front matter
PII: S0167-8191(02)00163-1

1. Introduction

Monte Carlo (MC) methods can loosely be defined as numerical processes that consume random numbers. MC computations have in the past, and continue to, consume a large fraction of available high-performance computing cycles. One of the reasons for this is that it is easy to parallelize these computations to achieve linear speed-up, even when communication latency is high, since usually little communication is required in MC. We discuss this further in Section 2.

Since MC computations depend on random numbers, the results can be adversely affected by defects in the random number sequence used. A random number generator (RNG) used in a program to produce a random sequence is actually a deterministic algorithm which produces numbers that look random to an application, and hence is often referred to as a pseudorandom number generator (PRNG). That is, the application produces an answer similar to what it would have with a truly random sequence, typically from a uniform distribution on the unit interval. [1]

Each PRNG has a finite number of possible states, and hence the "random" sequence will start repeating after a certain "period," leading to non-randomness. Typically, sequences stop behaving like a truly random sequence much before the period is exhausted, since there can be correlations between different parts of the sequence. We shall describe these terms further, and discuss PRNGs and parallel PRNGs (PPRNGs) in greater detail in Section 2.1.

Many of the RNGs in use today were developed and tested when computational power was a fraction of that available today. With increases in the speed of computers, many more random numbers are now consumed in even ordinary MC computations, and the entire period of many older generators can be consumed in a few seconds. Tests on important applications have revealed defects of RNGs that were not apparent with smaller simulations [12,14,30,31]. Thus RNGs have to be subjected to much larger tests than before. Parallelism further complicates matters, and we need to verify the absence of correlation among the random numbers produced on different processors in a large, multiprocessor computation. There has, therefore, been much interest over the last decade in testing both parallel and sequential random number generators [2,3,5,9–12,19,29,33,34], both theoretically and empirically.

While the quality of the PRNG sequence is extremely important, the unfortunate fact is that important aspects of quality are hard to prove mathematically. Though there are theoretical results available in the literature regarding all the popular PRNGs, the ultimate test of PRNG quality is empirical. Empirical tests fall broadly into two categories (i) statistical tests and (ii) application-based tests.

Statistical tests compare some statistic obtained using a PRNG with what would have been obtained with truly random independent identically distributed (IID) numbers on the unit interval. If their results are very different, then the PRNG is

[1] Non-uniform distributions can be obtained from a uniform distribution using certain standard techniques [18]. So we shall discuss only uniform distributions in the rest of the paper.

considered defective. A more precise explanation is given in Section 3.1. Statistical tests have an advantage over application-based tests in that they are typically much faster. Hence they permit the testing of a much larger set of random numbers than application-based tests. Certain statistical tests [16,21] have become de-facto standards for sequential PRNGs, and PRNGs that "pass" these tests are considered "good." We wish to note that passing an empirical test does not prove that the PRNG is really good. However, if a PRNG passes several tests, then our confidence in it increases. We shall later describe parallel versions of these standard tests that check for correlations in a PPRNG.

It turns out that applications interact with PRNGs in unpredictable ways. Thus, statistical tests and theoretical results are not adequate to demonstrate the quality of a PRNG. For example, the 32-bit linear congruential PRNG `CONG`, which is much maligned for its well known defects, performed better than the shift register sequence `R250` in an Ising model application with the Wolff algorithm [12], though the latter performed better with the Metropolis algorithm. Subsequently, defects in the latter generator were also discovered. Thus we need to test PRNGs in a manner similar to that of the application in which it is to be used. Typically, the application-based tests use random numbers in a manner similar to those of popular applications, except that the exact solution to the test applications are known. Such tests are described in Section 3.2.

When dealing with a new application, the safest approach is to run the application with different PRNGs. If the runs give similar results, then the answers can be accepted. The extra effort is not wasted, because the results from the different runs can be combined to reduce the statistical error.

Parallelism further complicates matters, and many users resort to ad hoc methods of PRNG parallelization. We later demonstrate defects in some of these strategies in Section 4. In order to avoid many of these pitfalls, the `SPRNG` [2] PPRNG library was developed. `SPRNG` provides a standard interface that permits users to easily change PRNGs and rerun their application, thus ensuring that the results are PRNG independent. These generators have also been subjected to some of the largest empirical tests of PRNGs, and correct defects in some popular generators. These test results too are presented in Section 4. The `SPRNG` software also comes with a suite of "standard" tests for PPRNGs, and can thus be used to also test non-`SPRNG` PRNGs.

The outline of the rest of the paper is as follows. In Section 2, we discuss parallelization of MC simulations, parallelization of PRNGs, and also mention some specific PPRNGs that we use in subsequent tests. We then describe empirical tests for PPRNGs, both statistical and application-based (physical-model tests), in Section 3, along with a description of their implementation in the `SPRNG` test suite. We presents test results in Section 4 and conclude with a summary of the results and recommendations on PPRNGs in Section 5.

[2] The `SPRNG` [32,26] parallel pseudorandom number library comes with a suite of tests to verify the quality of the generators, and is available at `http://sprng.cs.fsu.edu`.

2. Monte Carlo parallelization

One of the most common methods of MC parallelization is to use the same MC algorithm on each processor, and use a different random number stream on each processor. Results differ on the different processors due to differences in the random number sequences alone. These results are then combined to produce the desired answer with an overall smaller error than non-combined results, as shown in Fig. 1.

Such a parallelization scheme requires little communication between processors, and thus one can easily obtain a linear speed-up. This is the main reason for the popularity of MC on parallel computers.

Of course, the random number (RN) stream on each processors should be of high quality. In addition, there should be no correlation between RN streams on different processors. To illustrate the complications that can arise from parallelism, we consider the following extreme case. If the RN streams on all the processors were identical, then the results will be identical across the processors, and there would be no benefit from the parallelization. In real situations, there could be correlations between the RN streams across the different processors, or the streams could overlap, reducing the effectiveness of the parallelization. In the worst case, this could even lead to erroneous results, as demonstrated in a practical situation in Fig. 8 presented later, in Section 4 of this paper.

There is, therefore, a need to check for two types of defects in PPRNGs. First, we must verify that each RN stream on each processors is random. That is, there should be no apparent correlation between the elements within a single RN sequence. We refer to this as the absence of intra-stream correlations. Secondly, there should be no correlation between streams on the different processors. We call this the absence of inter-stream correlations.

We observe here that with the above method of parallelization, where the same algorithm was replicated on each processor, intra-stream correlations generally tend to affect the results more than inter-stream correlations of MC simulations of Markov Chains, which are probably the largest consumers of MC parallel computing cycles. There is an alternate parallelization using domain decomposition, where the computation of a single sequential run is distributed across the processors by dividing the state space across the processors. This is typically done when the solution space is very large. In that case, the initialization time could be significant, since in MC calculations with Markov Chain, the results of the first several steps are often

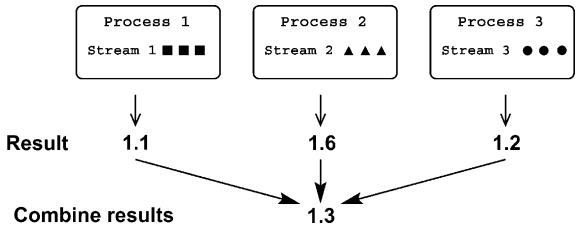

Fig. 1. Parallelization of MC computations through process replication.

discarded in order to prevent the results from depending on the initial state. In these computations, inter-stream and intra-stream correlation can have an equal effect on the solution. Thus, for the purpose of testing PPRNGs, such a decomposition is more effective.

2.1. Parallelizing PRNGs

We shall next describe popular methods of designing PPRNGs. Before doing this we explain some terms regarding sequential PRNGs, in terms of Fig. 2. A PRNG consists of a finite set of states, and a transition function T that takes the PRNG from one state to the next. The initial state of the PRNG is called the *seed*. Given a state S_i for the PRNG, there is a function F that can give a corresponding integer random number I_i or a floating point random number U_i. The memory used to store the state of a PRNG is a constant, and therefore the state space is finite. Therefore, if the PRNG is run long enough, then it will enter a cycle where the states start repeating. The length of this cycle is called the period of the PRNG. Clearly, it is important to have a large period.

We next discuss some common methods of PPRNG. In the methods of *cycle division*, a cycle corresponding to a single RNG is partitioned among the different processors so that each processor gets a different portion of the cycle (see [13,20], for example). This partitioning is generally performed in one of the following three ways. First, users randomly select a different seed on each processor and hope that the seeds will take them to widely separated portions of the cycle, so that there will be no overlap between the RN streams used by the different processors. In the second, *sequence splitting* scheme, the user deterministically chooses widely separated seeds for each processor. The danger here is that if the user happens to consume more random numbers than expected, then the streams on different processors could overlap. Apart from this, generators often have long-range correlations [7,8]. These long-range correlations in the sequential generator become short-range inter-stream correlations in such a parallel generator. Lastly, if there are n processors, then each stream in the *leap frog* scheme gets numbers that are n positions apart in the original sequence. For example, processor 0 gets random numbers $x_0, x_n, x_{2n}, \ldots$ This again

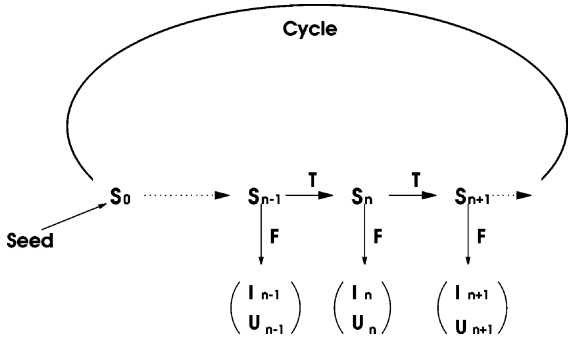

Fig. 2. This figure shows different terms associated with the state-space of a PRNG.

has problems because long range correlations in the original sequence can become short-range *intra-stream* correlations, which are often worse than inter-stream correlations.

Apart from these defects, the method of cycle-division results in a non-scalable period. That is, the number of different random numbers that can be used stays fixed, instead of increasing with the number of processors as in the scheme that is described below. In addition, since most PRNGs use modular arithmetic, the cost of generating RNs is dependent on the modulus chosen. In addition, the period of most PRNGs is also dependent on the modulus. Thus, with cycle division, longer periods are required as more numbers are generated on ever faster processors. This leads to the use of larger arithmetic moduli and a subsequent increase in the generation cost of individual RNs. Thus cycle-division is not a scalable procedure. In addition, we shall later give results that demonstrate statistical defects that arise in cycle-division-based schemes.

The parallelization scheme we recommend is based on *parameterization* (see [23–25,28], for example). This provides independent full-period streams on each processor. We can parameterize a set of streams by a *stream number*. Given the stream number i, there is an easy way of producing the ith stream. These parameterizations are done in two ways. The first is *seed parameterization:* in certain generators, the set of possible states 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 processors a seed from a different cycle. The other is *iteration function parameterization:* the iteration function is the function that gives the next state in the sequence, given the current state. In this method of 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 easily produced.

2.2. Parallel PRNGs tested

We next define the PPRNGs tested in this paper, and the method of their parallelization.

`SPRNG` *PPRNGs*: The following PPRNGs from the `SPRNG` libraries were tested:

1. *Combined multiple-recursive generator:* `cmrg`
   This generator is defined by the following recursion:

   $$z_n = x_n + y_n \times 2^{32} \pmod{2^{64}},$$

   where x_n is the sequence generated by the 64-bit linear congruential generator (LCG) given below and y_n is the sequence generated by the following prime modulus multiple-recursive generator (MRG):

   $$y_n = 107374182 y_{n-1} + 104480 y_{n-5} \pmod{2147483647}.$$

   The same prime modulus generator is used for all the streams. Streams differ due to differences in the 64-bit LCG. The period of this generator is around 2^{219}, and the number of distinct streams available is over 2^{24}.

2. *48-Bit linear congruential generator with prime addend:* `lcg`
   The recurrence relation for the sequence of random numbers produced by this generator is given by the following recurrence:

   $$x_n = ax_{n-1} + p \pmod{2^{48}},$$

   where p is a prime number and a is the multiplier. Different random number streams are obtained by choosing different prime numbers as the addend p [15,27]. The period of this generator is 2^{48}, and the number of distinct streams available is about 2^{19}.
3. *64-Bit linear congruential generator with prime addend:* `lcg64`
   The features of this generator are similar to the 48-bit LCG, except that the arithmetic is modulo 2^{64}. The period of this generator is 2^{64}, and the number of distinct streams available is over 2^{24}.
4. *Modified lagged-Fibonacci generator:* `lfg`
   The recurrence relation for this sequence of random numbers is given by the following equation:

   $$z_n = x_n \oplus y_n,$$

   where $\oplus$ is the exclusive-or operator, x and y are sequences obtained from additive lagged-Fibonacci generator (LFG) sequences X and Y of the following form:

   $$X_n = X_{n-k} + X_{n-\ell} \pmod{2^{32}},$$

   $$Y_n = Y_{n-k} + Y_{n-\ell} \pmod{2^{32}}.$$

   ℓ and k are called the lags of the generator, and we use the convention that $\ell > k$. x is obtained from X by setting the least-significant bit of the latter to 0. y is obtained from Y by shifting the latter right by one bit. This modification of the LFG is performed in order to avoid certain correlations that are observed in the unmodified generator.
   The period of this generator is $2^{31}(2^{\ell} - 1)$, and the number of distinct streams available is $2^{31(\ell-1)}$, where ℓ is the lag. For the default generator with lag $\ell = 1279$, the period is $\approx 2^{1310}$, and the number of distinct streams is 2^{39618}.
   The sequence obtained is determined by the ℓ initial values of the sequences X and Y. The seeding algorithm ensures that distinct streams are obtained during parallelization [23,24,28].
5. *Multiplicative lagged-Fibonacci generator:* `mlfg`
   The recurrence relation for this sequence of random numbers is given by the following equation:

   $$x_n = x_{n-k} \times x_{n-\ell} \pmod{2^{64}}.$$

   The period of this generator is $2^{61}(2^{\ell} - 1)$, and the number of distinct streams available is $2^{61(\ell-1)}$, where ℓ is the larger lag. For the default generator with lag $\ell = 17$, the period is approximately 2^{78}, and the number of distinct stream is 2^{976}.
   The sequence obtained is determined by the ℓ initial values of the sequence x. The seeding algorithms ensures that distinct streams are obtained during parallelization [25].

6. *Prime modulus linear congruential generator:* `pmlcg`
   This generator is defined by the following relation:

   $$x_n = ax_{n-1} \pmod{2^{61} - 1},$$

   where the multiplier a differs for each stream [22]. The multiplier is chosen to be certain powers of 37 that give maximal period cycles of acceptable quality. The period of this generator is $2^{61} - 2$, and the number of distinct streams available is roughly 2^{58}.

Each of these PPRNGs has several "variants." For example, changing the lags for the LFG will give a different variant of this generator. The user is not allowed to select an arbitrary parameter to get a different variant, but must choose from a set of well-tested ones. This is done by setting the `parm` argument in the initialization call for the PPRNG. This argument can always be set to `0` to get the default variant.

Other PRNGs:

1. `rand`: This is a 32-bit LCG available on Unix systems. We use it as a sequential PRNG to demonstrate defects even in its use on a single processor, as every PPRNG also needs to be a good serial PRNG.
2. `random`: The popular Unix additive LFG, `random`, with a lag of 31, was used to demonstrate defects even on a single processor. The `SPRNG` LFG corrects these defects.
3. `ranf`: The 48-bit Cray LCG, `ranf` is similar to the sequential version of the 48-bit LCG in `SPRNG` (as is the popular Unix generator `drand48`). We show that when using a sequence splitting scheme with this generator it fails a particularly effective test. The sequence splitting was performed by splitting the cycle into fragments of size 3×2^{35}, and using a different fragment for each segment. We chose a factor of 3 in the above fragment size because it is known that powers of two alone will result in strong correlations in this generator. In order to give the parallelization a fair chance of demonstrating its utility, we did not choose such an a priori worst-case situation.

3. Description of tests

A good PPRNG must also be a good sequential generator. Since the tests previously performed on sequential generators were not sufficiently large, we have performed much larger tests. Sequential tests check for correlations within a stream, while parallel tests check for correlations between different streams. Furthermore, applications usually require not just the absence of correlations in one dimension, but in higher dimensions as well. In fact, many simulations require the absence of correlations for a large number (say thousands) of dimensions.

PRNG tests, both sequential and parallel, can be broadly classified into two categories: (i) statistical tests and (ii) application-based tests. The basic idea behind sta-

tistical tests is that the random number streams obtained from a generator should have the properties of IID random samples drawn from the uniform distribution. Tests are designed so that the distribution of some test statistic is known exactly or asymptotically for the uniform distribution. The empirically generated RN stream is then subjected to the same test, and the statistic obtained is compared against the known distribution. While a boundless number of tests can be constructed, certain tests have become popular and are accepted as *de facto* standards. These include the series of tests proposed by Knuth [17], and the DIEHARD tests implemented by Marsaglia [21]. Generators that pass these tests are considered good.

It is also necessary to verify the quality of a PRNG by using it in real applications. Thus we also include tests based on physical models, which use random numbers in a manner similar to that seen in a real application, except that the exact solution is known. The advantage of the statistical tests is that these tests are usually much faster than the application-based ones. On the other hand, the latter use random numbers in the same manner as real applications, and can thus be considered more representative of real random number usage, and also typically test for correlations of more numbers at a time.

3.1. Statistical tests

We can modify sequential RN tests to test PPRNGs by interleaving different streams to produce a new RN stream. This new stream is then subjected to the standard sequential tests. For example if stream i is given by $x_{i0}, x_{i1}, \ldots$, $0 \leqslant i < N$, then the new stream is given by $x_{00}, x_{10}, \ldots, x_{N-1,0}, x_{01}, x_{11}, \ldots$ If each of the individual streams is random, and the streams are independent of each other, then the newly formed stream should also be random. On the other hand, any correlation between streams manifests as intra-stream correlation in the interleaved stream, and this can be tested with the conventional PRNG tests.

We form several such new streams, and test several blocks of random numbers from each stream. Usually the result of the test for each block is a Chi-square value. We take the Chi-square statistics for all the blocks and use the Kolmogorov–Smirnov (KS) test to verify that they are distributed according to the Chi-square distribution. If the KS percentile is between 2.5% and 97.5%, then the test is passed by the RN generator. The `SPRNG` [32] test suite provides a standard implementation to perform these tests.

In presenting the test result below, we shall let `ncombine` denote the number of streams we interleave to form a new stream, `nstreams` the number of new streams, and `nblocks` the number of blocks of random numbers from each new stream tested. In order to permit reproduction of our test results, we give a few more arguments that are specific to the `SPRNG` implementation: `seed` denotes the (encoded) seed to the RN generator, `param` the parameter for the generator, and `skip` the number of random numbers we skip after testing a block, before we start a test on the next block.

We next briefly describe each test followed by its test specific arguments. We also give the number of RNs tested and asymptotic memory requirements (in bytes,

assuming an integer is 4 bytes and a double precision floating point number is 8 bytes).

The details concerning these tests are presented in Knuth [17], unless we mention otherwise. The SPRNG 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 $\log d$ most-significant bits from $\log md$ random integers to form a new $\log m = \log md * \log d$-bit random integer. We form $n \ll m$ such numbers. A collision is said to have occurred each time some such number repeats. We count the number of collisions and compare with the expected number. This test thus checks for absence of $\log d$-dimensional correlation. It is one of the most effective tests among those proposed by Knuth.

Number of RNs tested: $n * \log md$

Memory: $8 * nstreams * nblocks + 4 * n + 2^{\log md * \log d}$

2. *Coupon collector's test: n t d*

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

Number of RNs tested: $n * d * \log d$

Memory: $8 * nstreams * nblocks + 4 * d + 16 * (t - d + 1)$

3. *Equidistribution test: d n*

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

Number of random numbers tested: n

Memory: $8 * nstreams * nblocks + 16 * d$

4. *Gap test: t a b n*

We generate floating point numbers in (0,1) and note the gap in the sequence between successive appearances of numbers in the interval $[a, b]$ in (0,1). For example, if $[a, b] = [0.4, 0.7]$ and the sequence is: $0.1, 0.5, 0.6, 0.9, \ldots$, 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.

Number of RNs tested: $n/(b - a)$

Memory: $8 * nstreams * nblocks + 16 * t$

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.

Number of RNs tested: $n * m$

Memory: $8 * nstreams * nblocks + 16 * n$

6. *Permutations test: m n*

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

Number of RNs tested: $n * m$

Memory: $8 * nstreams * nblocks + 8 * m + 16 * (m!)$

7. *Poker test: n k d*

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

Number of RNs tested: $n * k$

Memory: $8 * nstreams * nblocks + 0.4 * \min(n, k) + 12 * k + 4 * d$

8. *Runs up test: t n*

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

Number of RNs tested: $1.5 * n$

Memory: $8 * nstreams * nblocks + 16 * t$

9. *Serial test: d n*

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

Number of RNs tested: $2 * n$

Memory: $8 * nstreams * nblocks + 16 * d * d$

There are certain other tests that are inherently parallel, in contrast to the above scheme which is really parallelization of sequential tests. Since these tests are inherently parallel, we need not interleave streams, and thus *ncombine* = 1, while *nstreams*, is the total number of streams tested. All these streams are tested simultaneously, rather than independently as in the previous case. We describe the tests below.

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 RN 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. The `SPRNG` test suite implementation uses the normality assumption in the percentile given as the result. However, it also gives a range where the exact percentile will lie. Thus users are given a margin of error.)

2. *Fourier transform test: n*

We fill a two-dimensional array with RNs. Each row of the array is filled with n RNs from a different stream. We calculate the two-dimensional Fourier coefficients and compare with the expected values. This test is repeated several times and we check if there are particular coefficients that are repeatedly "bad." If the same coefficients turn out to be high in repeated runs, or if the number of coefficients that are high is much more than expected, then we can suspect defects in the generators. This test does not give a single number that can be presented as a result, and so we shall not mention it in Section 4. We did small tests on the SPRNG generators and did not detect any anomalies.

3.2. Application-based tests

Application-based tests use random numbers in a manner similar in which they are used in practical applications. Generally, the exact behavior of the test is known analytically. We describe the tests implemented in the SPRNG test suite.

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 [1]. 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 in testing random number generators, and has often detected subtle defects in generators [2,3,12,31,35].

We can test parallel generators on the Ising model application by assigning distinct random number sequences to different subsets of lattice sites [3]. This is essentially the domain decomposition method of MC parallelization, and, as mentioned earlier, 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 larger number of streams simultaneously.

We next describe the implementation of these tests in the SPRNG test suite. The user selects a lattice size, a seed to the RNG, and the variant of the RNG as command line arguments. Since the configurations change only slightly at successive times steps, it is necessary to average the results over a larger block. The size of these blocks too is specified by the user. The user also specifies the number of such blocks whose results need to be computed. The initial state is chosen by assigning a random spin to each lattice site. In order to prevent the results from being influenced by this particular choice of initial state, the user also needs to specify the number of initial blocks to be discarded. The tests are carried out with $J/K_bT = 0.4406868$, where J is the energy per bond and T is the temperature. This parameter can be changed in the code, to run the tests at different temperatures.

In order to test the quality of the generator, we plot absolute error versus standard deviation for the specific heat and energy at different points in the simulation. The points should fall below the $2-\sigma$ line (the line corresponding to twice the stan-

dard deviation) most of the time. In a bad generator, as the standard deviation decreases (with increase in sample size), the error does not decrease as fast and remains above this line. An example of both cases is demonstrated later in Fig. 8.

2. *Random walk test:* The random walk test implemented is a simple one, based on a previously described algorithm [35]. 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 RN 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 statistical tests. PPRNGs are tested by interleaving streams.

4. Test results

In this section, we first demonstrate defects in some popular sequential generators, and in cycle-division strategies for parallelization of PRNGs. We then present detailed results of tests on two `SPRNG` generators (`lcg`, which had problems, and `mflg`, which did not) and summarize test results for the other generators.

As mentioned earlier, the results of the statistical tests are considered passed if the KS percentile is between 2.5% and 97.5%. However, this implies that even a good generator will "fail" around 5% of the tests. Therefore, when we observe a percentile close to (on either side) of the pass thresholds, we repeated the calculations with different seeds. A good generator is expected to give percentiles in different ranges with different seeds, while a bad one would consistently fail. In most of the calculations, we take the (global) seed [3] to be 0. The above calculations are generally the reason for some of the seeds displayed being different from this. We sometimes mention that a PRNG passed a test with N numbers. This means that it passed a test, with the total number of RNs used in the test being around N.

4.1. Tests on sequential generators

The popular 32 bit Unix LCG, `rand` fails even sequential PRNG tests. [4] For example, the collisions test fails for the following parameters: $d = 4$, $k = 5$, $n = 2 \times 10^5$, when this test was repeated 40 times. The K–S percentile is obtained as 100.00. It keeps failing this test even with different seeds. Thus this generator fails with about 10^7–10^8 random numbers. On a machine capable of generating 10^7 RNs

[3] The term *seed* is typically used for the starting state of the generator. However, in order to simplify the interface, `SPRNG` uses a global seed, which is a 31 bit integer. Based on this global seed, and on the stream number, the actual state is produced. In the rest of this document, we shall use the term "seed" to refer to this global seed.

[4] Note that the `rand` implementation on recent Linux systems is different; it is the same generator as the usual PRNG `random`.

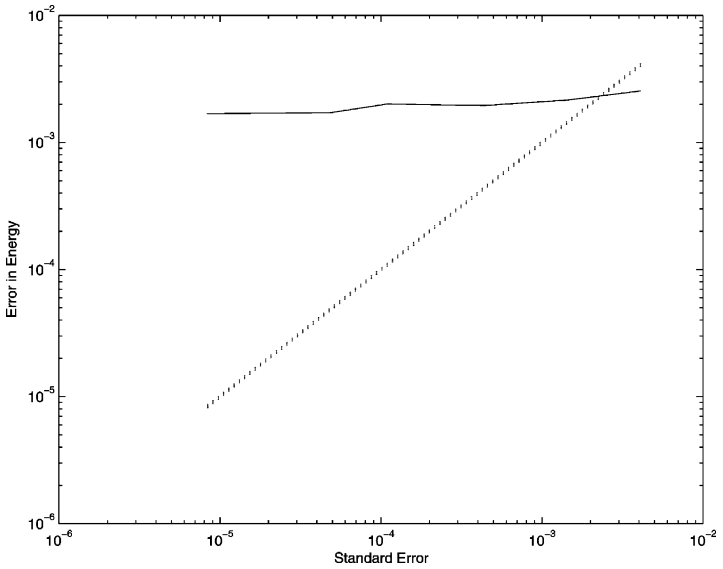

Fig. 3. Plot of the actual error versus the internally estimated standard deviation of the energy error for Ising model simulations with the Wolff algorithm on a 16×16 lattice with `random`, an additive LFG (solid line). The dotted line is an error equal to the standard deviation.

per second, this would take between 1 and 10 s! Therefore this generator should not be used any longer.

The popular Unix additive LFG, `random`, with a lag of 31, fails even the sequential gap test with the following parameters: $[a, b] = [0.5, 0.51]$, gaps of length > 200 are grouped together in one category, 10^7 gap lengths were noted, and the test was repeated 100 times for a total of around 10^{11} RNs. The Birthday Spacings test in the DIEHARD package is more effective and detects the defects in additive LFGs with much fewer RNs. This generator also fails sequential application-based tests. For example, the Wolff algorithm fails with around 10^8 random numbers for a 16×16 lattice, as shown in Fig. 3.

The `SPRNG` LFG, in contrast, combines two different RN streams to produce a new stream. This generator passes the gap test even with 10^{13} random numbers used, which is one of the largest tests ever of random numbers. [5] It also passes the Ising model tests, both sequential and parallel, with around 10^{11} RNs.

4.2. Tests with cycle-division

Among cycle division strategies, sequence-splitting is considered the most effective [4]. However, there are theoretical results suggesting weaknesses in cycle-division strategies. We demonstrate these defects empirically, using sequence-splitting.

[5] The test was run on the CONDOR facility at the University of Wisconsin at Madison.

The 48-bit Cray LCG, `ranf` is similar to the sequential version of the 48-bit LCG in `SPRNG` (as is the popular Unix generator `drand48`). We show that using a sequence splitting scheme with this generator fails the blocking test with around 10^{10} RNs. The blocking test was performed with 128 streams, with 256 random numbers from each stream being added together to form a sum, and 10^6 such sums being generated. The sequence splitting was performed by splitting the sequence into fragments as mentioned earlier. The generator failed this test, giving a K–S test percentile of 100.00, demonstrating the danger of cycle-division. We observe that it is popularly believed that the main problem with sequence-splitting is the possibility of overlap in the sequences, which can be avoided by ensuring that each subsequence is sufficiently large. However, in our test, no overlap occurs. The defect is solely due to long range correlations. Periodically, theoretical results are published exposing previously undetected long range correlation in some generators [7,8]. Thus one needs to be wary of using cycle-division-based parallelization.

4.3. Tests on the `SPRNG` multiplicative LFG

We give the results of the tests on the `SPRNG` multiplicative lagged Fibonacci generator (MLFG). This generator gives good results even with small lags, such as 17,5.

4.3.1. Sequential tests

We first give the results of tests on a sequential version of this generator.

The DIEHARD [21] test suite runs the following tests to verify the randomness of a block of approximately three million RNs. The tests in DIEHARD are the Birthday Spacings test, the Overlapping 5-Permutation Test, Binary Rank Test (for 6×8, 31×31 and 32×32 matrices), Bitstream Test, the Overlapping-Pairs-Sparse-Occupancy (OPSO) Test, the Overlapping-Quadruples-Sparse-Occupancy (OQSO) Test, the DNA Test, the Count-The-1's Test (for a stream of bytes and for specific bytes), the Parking Lot Test, the Minimum Distance Test, the 3DSpheres Test, the Squeeze Test, the Overlapping Sums Test, Runs Test, and the Craps Test. All the tests in the DIEHARD suite pass, even with the small lag of 17.

The first 1024 streams of each of the variants of the MLFG were tested with the sequential statistical tests from the `SPRNG` test suite, and with the random walk test. The parameters were chosen to test around 10^{11} RNs, except for the collisions test, which used 10^{12} RNs. The latter test was larger since that test is particularly effective in detecting defects. The random walk test used just around 10^8 RNs, since we have implemented a rather simple version. The details of the parameters are summarized in Table 1, indicating that all the tests passed.

The 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 1 000 000 blocks were considered, after discarding the results of the first 100 blocks. This tests over 10^{11} RNs. The generator passed both these tests as shown in Figs. 4 and 5. These figures show the plot for the specific heat with the Metropolis algorithm,

Table 1
Sequential PRNG tests on `mlfg`

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

The common test parameters 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 lags 17,5, thus with `param` = 0.

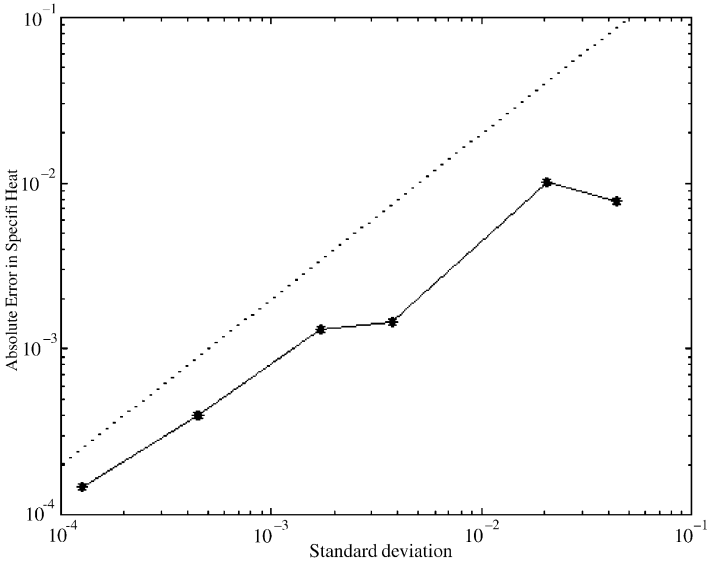

Fig. 4. Plot of the actual error versus the internally estimated standard deviation of the specific heat for Ising model simulations with the Metropolis algorithm on a 16×16 lattice with a sequential version of the `SPRNG mlfg` with lags (17,5). We expect around 95% of the points to be below the dotted line (representing an error of two standard deviations) with a good generator.

and with the energy for the Wolff algorithm. We actually tested both energy and specific heat for both the algorithms.

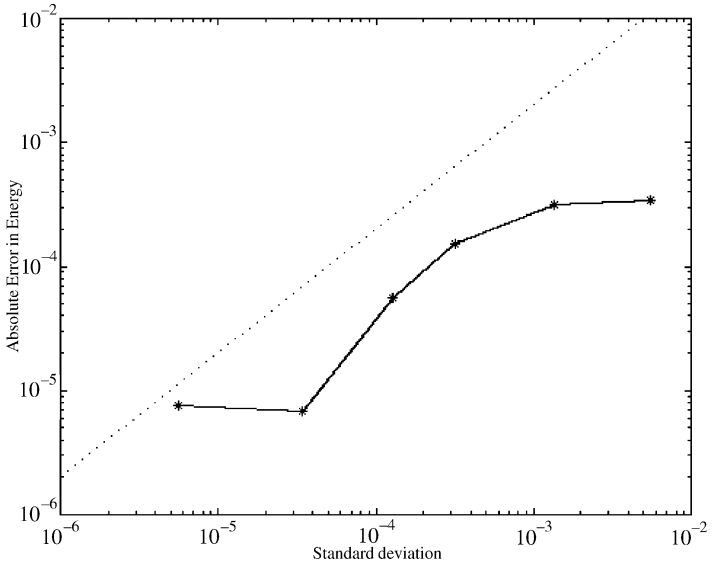

Fig. 5. Plot of the actual error versus the internally estimated standard deviation of the specific heat for Ising model simulations with the Wolff algorithm on a 16×16 lattice with a sequential version of the SPRNG `mlfg` with lags (17,5). We expect around 95% of the points to be below the dotted line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

4.3.2. Parallel tests

In contrast to the SPRNG generator `lcg` (described later), the SPRNG generator `mlfg` passed the tests without requiring any modifications, even for small lags. For those tests from the 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 passed. The number of RNs consumed in each test was around 10^{11}, unless mentioned otherwise. As with the sequential tests, we used around 10^{12} RNs 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 over the sequential test.

The 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 1 000 000 blocks were considered, after discarding the results of the first 100 blocks. This tests over 10^{11} RN. Fig. 6 shows the plot for the specific heat with the Metropolis algorithm.

Note: In contrast to the MLFG, a plain parameterized LFG fails the parallel gap test with as few as 10^6 RNs [25], even when a lag as large as 1279 is used. The sequential gap test is passed with this lag even when the number of random numbers used is around 10^{11}, demonstrating both, the presence of inter-stream correlations, and the effectiveness of interleaving in detecting these correlations.

Table 2
Parallel PRNG tests on `mlfg`

Test	Parameters	K–S percentile
Blocking	$n = 1\,000\,000$, $r = 128$	10.2
Collision	$n = 200\,000$, $\log md = 10$, $\log d = 3$	60.6
Collision	$n = 200\,000$, $\log md = 4$, $\log d = 5$	46.8
Coupon	$n = 5\,000\,000$, $t = 30$, $d = 10$	12.9
Gap	$t = 200$, $a = 0.5$, $b = 0.51$, $n = 1\,000\,000$	78.0
Permutations	$m = 5$, $n = 20\,000\,000$	7.2
Poker	$n = 10\,000\,000$, $k = 10$, $d = 10$	9.7
Random walk	Walk length = 1024	6.8
Runs	$t = 10$, $n = 50\,000\,000$	33.2
Serial	$d = 100$, $n = 50\,000\,000$	31.2

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 above the values `nstreams` = 32, `ncombine` = 32, and `nblock` = 160 were used. While in the second one the values `nstreams` = 16, `ncombine` = 64, and `nblocks` = 50 000 were used. (iii) The random walk test used `nblocks` = 25 000 for around 10^8 RNs. The K–S test percentile given above is for the default lags 17,5, thus with *param* = 0.

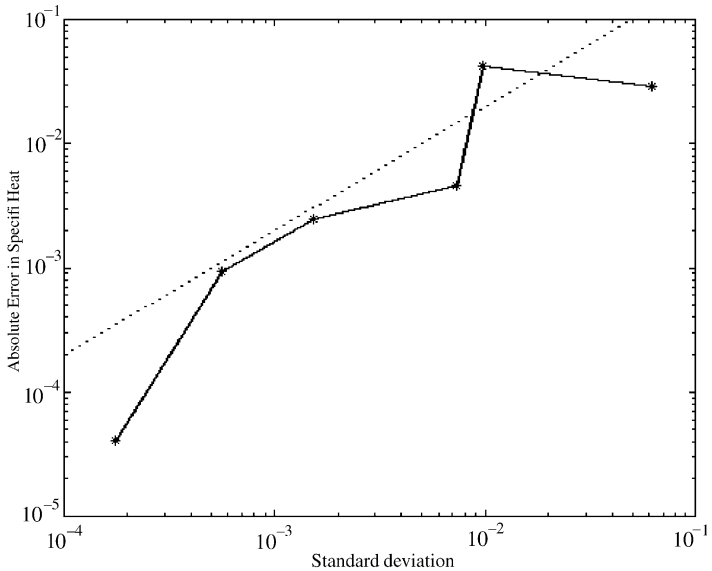

Fig. 6. Plot of the actual error versus the internally estimated standard deviation of the specific heat for Ising model simulations with the Metropolis algorithm on a 16×16 lattice with a sequential version of the `SPRNG` generator `mlfg` with lags (17,5). We expect around 95% of the points to be below the dotted line (representing an error of two standard deviations) with a good generator.

4.4. Tests on LCG

We give details of the tests on the SPRNG lcg generator, which is a power-of-two modulus 48-bit LCG. The defects in power-of-two modulus generators are well known, and so this generator should typically not be used as the default. However, it can be used to verify the results of computations first performed using another generator. If the two results agree, then the results from both the generators can be combined to reduce the error estimate. We also wish to note that the 48-bit and 64-bit generators perform well in most real applications, as long the the application is not sensitive to correlations between RNs that are a power-of-two apart in the sequence. In contrast, the 32-bit version should probably never be used.

4.4.1. Sequential tests

We first give the results of tests on a sequential version of this generator.

All the DIEHARD tests passed, except that the lower order bits (generally the 8–10 lowest bits) failed the DNA, OQSO, and OPSO tests. The poorer quality of the lower bits is expected from theoretical results. SPRNG include a 64-bit LCG too, to ameliorate this defect. In addition, SPRNG includes a combined multiple recursive generator (CMRG) that combines a 64-bit LCG stream with a stream from a MRG to produce a better stream. Even the lower order bits of these generators pass all the DIEHARD tests. [6]

The first 1024 streams of each of the variants of the 48-bit LCG were tested with the sequential statistical tests from SPRNG test suite, and with the random walk test. The parameters were chosen to test around 10^{11} RNs, except that the collisions test used 10^{12} RNs and the random walk test used just around 10^{8} RNs, for the same reason as with the MLFG tests. The details of the parameters are summarized in Table 3, indicating that all the tests passed.

The 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 1 000 000 blocks were considered, after discarding the results of the first 100 blocks. This tests over 10^{11} RN. The generator passed both these tests.

4.4.2. Parallel tests

We now give the results of tests on the parallel version of this generator. The parallelization is through parameterization, as mentioned in Section 2.1.

In the original version of this generator, all the streams were started from the same initial state. The streams differed due to differences in the additive constants.

[6] Note that the least-significant bits of the states of the 64-bit LCG use in the CMRG will have the same defects as that of the 48-bit LCG. However, when we generate a RN from the state of the LCG, we use only the most-significant bits. For example, a random integer will use the 32 most-significant bits of the state. Thus the least-significant bit of a RN from the 64-bit LCG is the 33rd least-significant bit of the corresponding state, whereas in the 48-bit LCG, it is the 17th least-significant bit. Thus even the lower order bits of a RN from a 64-bit LCG can be expected to be much better than the corresponding ones from a 48-bit LCG, as proved by the tests.

Table 3
Sequential PRNG tests on the SPRNG generator lcg

Test	Parameters	K–S percentile
Collisions	$n = 100000$, $\log md = 10$, $\log d = 3$	63.7
Collision	$n = 200000$, $\log md = 4$, $\log d = 5$	73.8
Coupon	$n = 5000000$, $t = 30$, $d = 10$	67.2
Equidist	$d = 10000$, $n = 100000000$	3.1
Gap	$t = 200$, $a = 0.5$, $b = 0.51$, $n = 1000000$	6.5
Maxt	$n = 50000$, $t = 16$	62.6
Permutations	$m = 5$, $n = 20000000$	85.8
Poker	$n = 10000000$, $k = 10$, $d = 10$	17.8
Random walk	Walk length = 1024	21.2
Runs	$t = 10$, $n = 50000000$	80.7
Serial	$d = 100$, $n = 50000000$	73.9

The common test parameters 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 poker test seed was 9999. The K–S test percentile given above is for the default multiplier 2875*a2e7b*175 (base 16), thus with *param* = 0.

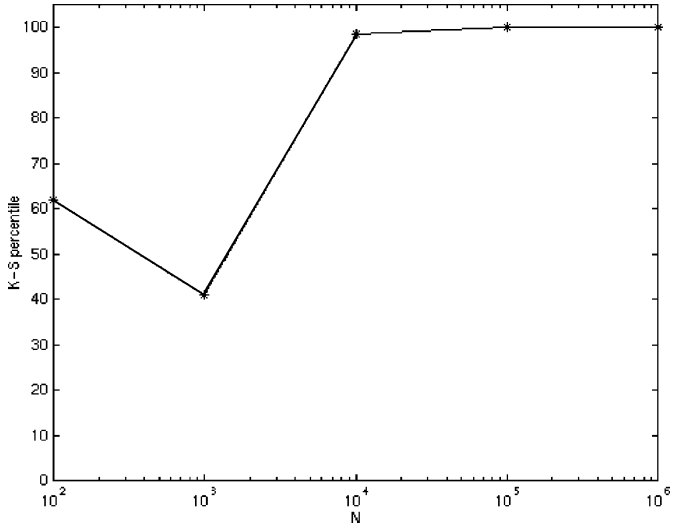

Fig. 7. Plot of the percentile from the K–S test versus the number of sums added together in the blocking test. The parameter ngroups was taken to be 128, and we used 1024 streams in this test. The PPRNG used was an older version of the SPRNG generator lcg, where each stream was started from the same initial state.

It can be seen from the blocking test results in Fig. 7 and the dashed line of Fig. 8 that this generator has inter-stream correlation. (Since the sequential tests passed, this failure could not be due to intra-stream correlations.)

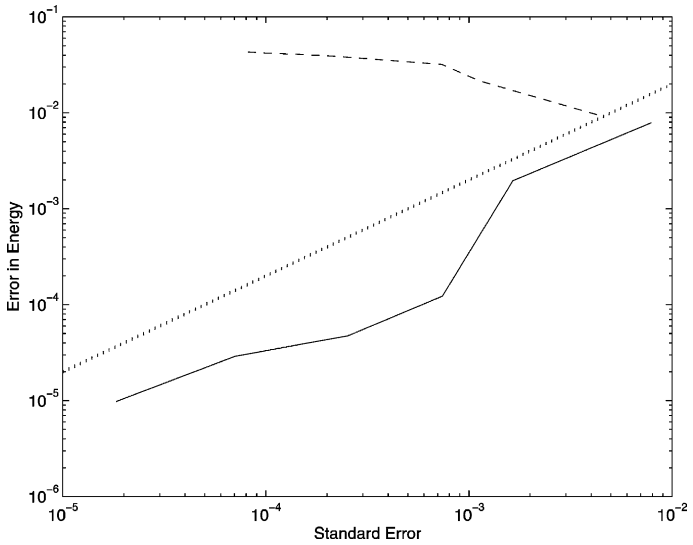

Fig. 8. Plot of the actual error versus the internally estimated standard deviation of the energy error for Ising model simulations with the Metropolis algorithm on a 16×16 lattice with a different LCG sequence from the `SPRNG lcg` generator used at each lattice site. The dashed line shows the results when all the LCG sequences were started with the same seeds but with different additive constants. The solid line shows the results when the sequences were started with different seeds. We expect around 95% of the points to be below the dotted line (which represents an error of two standard deviations) with a good generator.

Even if we discard the first million numbers from each sequence, these correlations persist and the streams still fail these tests. In the final version of this generator, the initial states are staggered so that each stream starts at a sufficiently different position in the sequence. Then, the tests are passed, as shown by the solid line of Fig. 8 and from the results in Table 4. Thus one needs to be careful with the seeding, even with parallelization through the iteration function.

For those tests from the 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 4, indicating that all the tests passed. The number of RNs consumed in each test was around 10^{11} except for the Gap, Runs, and Collisions tests, which used 10^{12}, and for the random walk test which used just 10^{8}.

The 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 1 000 000 blocks were considered, after discarding the results of the first 100 blocks. This tests over 10^{11} RN. Fig. 8 shows the plot for the energy with the Metropolis algorithm.

4.4.3. Miscellaneous results

De'Matteis and Pagnutti [6] showed that power-of-two modulus LCGs parallelized through additive constants differ only by a constant (modulo the modulus), if

Table 4
Parallel PRNG tests on the SPRNG generator lcg

Test	Parameters	K–S percentile
Blocking	$n = 1000000$, $r = 128$	73.4
Collision	$n = 200000$, $\log md = 4$, $\log d = 5$	9.9
Coupon	$n = 5000000$, $t = 30$, $d = 10$	55.4
Gap	$t = 200$, $a = 0.5$, $b = 0.51$, $n = 10000000$	25.3
Permutations	$m = 5$, $n = 20000000$	62.3
Poker	$n = 10000000$, $k = 10$, $d = 10$	87.7
Random walk	Walk length = 1024	5.6
Runs	$t = 10$, $n = 500000000$	64.4
Serial	$d = 100$, $n = 50000000$	60.3

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 used nstreams = 16, ncombine = 64, and nblocks = 50000. (iii) The *random walk* test used blocks = 10000, and seed = 9111999. The K–S test percentile given above is for the default multiplier 2875*a*2*e*7*b*175 (base 16), thus with param = 0.

the streams are shifted by a certain "shift-factor." More formally, given two sequences y and z such that

$$y_{n+1} = ay_n + c_1 \pmod{2^t} \text{ and } z_{n+1} = az_n + c_2 \pmod{2^t}$$

where a is the same maximal period multiplier in both cases, c_1 and c_2 are odd integers, and 2^t is the modulus, there exists an s and c such that

$$z_{n+s} = y_n + c \pmod{2^t} \tag{1}$$

We performed tests to determine the shift factor s required for the first 1000 streams produced by the LCG. If the shifts were much smaller than the size of the sequence used from each stream, then this could result in inter-stream correlations. Let us define the distance between two streams as the smallest shift required to satisfy Eq. (1). For each of the 1000 streams, we determined the distance to the nearest stream out of this set of streams. We give the results for the multiplier 2875*a*2*e*7*b*175 in base 16. The mean distance between streams was 7×10^{10}, the median was 4×10^{10}, the maximum distance was 5×10^{11} and the minimum 1×10^{8}. Note that in our larger parallel tests, we checked for a total of around 10^{12} RNs across 1024 streams, for around 10^{9} RNs per stream. Since this is smaller than the typical shift, the tests could not detect this correlation, as expected. This also demonstrates that the shifts with these particular streams are sufficiently large that they do not pose a problem in practice. We note, however, that if the number of streams is increased, then we can expect the nearest neighbors to come closer, and thus the mean shifts will decrease. Conversely, if the number of streams used is lower, or if the modulus is higher (as with the 64-bit generator), then the shifts will be higher.

4.5. Summary of tests results

We give below a summary of the results of tests on the SPRNG generators.

All the SPRNG generators were tested with the DIEHARD suite, and they all passed these tests, [7] except that the lower order bits of the 48-bit LCG are bad. The following tests from Knuth [17] were performed, including their parallel versions: collisions, coupon collector, equidistribution, gap, maximum of t, permutations, poker, runs-up, serial. At least 10^{11} RNs were tested for each generator in each case. The collisions test used 10^{12} RNs. The sequential gap test for the additive LFG used 10^{13} RNs—one of the largest empirical RN tests ever accomplished. The parallel gap test for this generator used 10^{12} RNs. We also performed the blocking test for parallel generators, and the Metropolis and Wolff algorithm for the Ising model with at least 10^{11} RNs. All SPRNG generators passed these tests. More details and the latest results can be found at the SPRNG web site.

5. Conclusions

In this section we shall describe the weaknesses of different generators and give general guidelines on their use. We also mention those tests that we also found to be particularly effective in detecting defects.

LCGs with power-of-two moduli are known to have extremely non-random lower order bits—the ith least-significant bits have a period of 2^i. Thus, if an application is sensitive to lower order bits, then erroneous results can be obtained. The larger the modulus, the farther are the least-significant bits from the most-significant bits, and thus we can expect these correlations to influence a floating point random number to a lesser extent. The 32-bit generator `rand` is extremely bad by today's standards, and should not be used, either in parallel or in sequential calculations. The 48-bit generator, `lcg`, is much better, and of course, the 64-bit generator, `lcg64`, is better yet. Despite this problem, `lcg` has proven to be good in most applications. However, one should take care to ensure that the application is not sensitive to this power-of-two correlation.

The combined multiple-recursive generator, `cmrg` attempts to remedy this problem by combining the 64-bit LCG with a multiple-recursive generator. We use a 32-bit version of `cmrg`, which improves the quality of the 32 most-significant bits. However, the lower order bits are still identical to those of the corresponding LCG.

The SPRNG prime modulus LCG `pmlcg` does not have these power-of-two correlations. However, some caution is required regarding the parallel generator. We use different multipliers to ensure that different streams are obtained. However, if

[7] All the final versions of the SPRNG generators passed. Original versions sometimes had defects which had to be corrected with, for example, `lfg`, `lcg`, and `pmlcg`. The cases of `lfg` and `lcg` have been mentioned above. In the case of `pmlcg`, we use different multipliers for parameterization. As one might expect, some of these multipliers are not as good as others. We therefore eliminated the defective ones, and tests using the first 1024 multipliers in the current implementation suggest that they are quite good.

we consider a large number of streams, there may be a few with bad multipliers. Furthermore, the initialization of this generator is a bit slow compared with the others.

The additive LFG implemented in the SPRNG package, `lcg`, actually combines two different RN streams to produce a new one. Otherwise, a plain additive LFG sequence fails certain tests such as the Birthday Spacings test implemented in DIEHARD, and the gap test described in Section 3. To be on the safe side, we suggest using a large lag. The multiplicative LFG, `mlfg`, is the only generator considered here that has a fundamentally non-linear recurrence relation. We expect it to be safe to use it even with a small lag.

Among the tests, the Birthday Spacings test (from DIEHARD), the Collision, Gap, and Runs tests (from Knuth, and their parallel versions as implemented in the SPRNG test suite), and the Blocking and Ising model tests were particularly effective in exposing defects in PRNGs and PPRNGs. One problem is that DIEHARD is "hardwired" to perform its suite of tests on a fixed number of RNs. This is somewhat restrictive. However, a general version of the DIEHARD suite, which allow the user to vary the number of RNs tested, has now been implemented, and should be available in future releases of SPRNG.

We conclude this paper by recommending the technique of independent streams for parallelizing RN generators. Using the technique of cycle division can turn long range correlations in the original sequence into short range inter- or intra-stream correlation. [8] The SPRNG software provides several generators using parameterization, and suite of "standard" parallel tests to test even non-SPRNG generators. In the future, we expect to provide a web-based testing facility as well.

Acknowledgements

The SPRNG software was developed with funding from DARPA Contract Number DABT63-95-C-0123 for ITO: Scalable Systems and Software, entitled *A Scalable Pseudorandom Number Generation Library for Parallel Monte Carlo Computations*. 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] P.D. Beale, Exact distribution of energies in the two-dimensional Ising model, Phys. Rev. Lett. 76 (1996) 78.

[8] There have been some warnings about the dangers of using independent streams [4] since it may so happen that different parameters just take you to different points on the same sequence. However, this is really only a warning against a naive parallelization strategy. One can design (as we have done in the SPRNG package) a parallelization strategy that avoids this pitfall.

[2] P. Coddington, Analysis of random number generators using Monte Carlo simulation, Int. J. Mod. Phys. C 5 (3) (1994) 547–560.
[3] P. Coddington, Tests of random number generators using Ising model simulations, Int. J. Mod. Phys. C 7 (3) (1996) 295–303.
[4] P. Coddington, Random number generators for parallel computers, 28 April 1997. Available at: www.npac.syr.edu/users/paulc/papers/NHSEreview1.1/PRNGreview.ps.
[5] S.A. Cuccaro, M. Mascagni, 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, Philadelphia, Pennsylvania, 1995, SIAM, pp. 279–284.
[6] A. De'Matteis, S. Pagnutti, A class of parallel random number generators, Parallel Comput. 13 (1990) 193–198.
[7] A. De'Matteis, S. Pagnutti, Long-range correlations in linear and non-linear random number generators, Parallel Comput. 14 (1990) 207–210.
[8] A. De'Matteis, S. Pagnutti, Parallelization of random number generators and long-range correlations, Parallel Comput. 15 (1990) 155–164.
[9] A. De'Matteis, S. Pagnutti, Controlling correlations in parallel Monte Carlo, Parallel Comput. 21 (1995) 73–84.
[10] M.J. Durst, Testing parallel random number generators, In: Computing Science and Statistics: Proceedings of the XXth Symposium on the Interface, 1988, pp. 228–231.
[11] K. Entacher, Bad subsequences of well-known linear congruential pseudorandom number generators, ACM Trans. Model. Comput. Simul. 8 (1998) 61–70.
[12] A.M. Ferrenberg, D.P. Landau, Y.J. Wong, Monte Carlo simulations: hidden errors from "good" random number generators, Phys. Rev. Let. 69 (1992) 3382–3384.
[13] P. Frederickson, R. Hiromoto, T.L. Jordan, B. Smith, T. Warnock, Pseudo-random trees in Monte Carlo, Parallel Comput. 1 (1984) 175–180.
[14] P. Grassberger, On correlations in 'good' random number generators, Phys. Lett. A 181 (1) (1993) 43–46.
[15] J.H. Halton, Pseudo-random trees: multiple independent sequence generators for parallel and branching computations, J. Comput. Phys. 84 (1989) 1–56.
[16] D.E. Knuth, The Art of Computer Programming, vol. 2: Seminumerical Algorithms, second ed., Addison-Wesley, Reading, MA, 1981.
[17] D.E. Knuth, The Art of Computer Programming, vol. 2: Seminumerical Algorithms, third ed., Addison-Wesley, Reading, MA, 1998.
[18] S.S. Lavenberg, Computer Performance Modeling Handbook, Academic Press, New York, 1983.
[19] P. L'Ecuyer, J.-F. Cordeau, R. Simard, Close-point spatial tests for random number generators, Operations Research 48 (2) (2000). Available at: http://www.iro.umontreal.ca/lecuyer/papers.html (npair.ps), 1999.
[20] J. Makino, Lagged-Fibonacci random number generator on parallel computers, Parallel Comput. 20 (1994) 1357–1367.
[21] G. Marsaglia, Diehard software package, Available at: ftp://stat.fsu.edu/pub/diehard.
[22] M. Mascagni, Parallel linear congruential generators with prime moduli, Parallel Comput. 24 (1998) 923–936 (and 1997 IMA Preprint #1470).
[23] M. Mascagni, S.A. Cuccaro, D.V. Pryor, M.L. Robinson, A fast, high-quality, and reproducible lagged-Fibonacci pseudorandom number generator, J. Comput. Phys. 15 (1995) 211–219.
[24] M. Mascagni, M.L. Robinson, D.V. Pryor, S.A. Cuccaro, Parallel pseudorandom number generation using additive lagged-Fibonacci recursions, Springer Verlag Lecture Notes in Statistics 106 (1995) 263–277.
[25] M. Mascagni, A. Srinivasan, Parameterizing parallel multiplicative lagged-Fibonacci generators, Parallel Comput., submitted for publication.
[26] M. Mascagni, A. Srinivasan, SPRNG: A scalable library for pseudorandom number generation, ACM Trans. Math. Software 26 (2000) 436–461.
[27] O.E. Percus, M.H. Kalos, Random number generators for MIMD parallel processors, J. Par. Distr. Comput. 6 (1989) 477–497.

[28] D.V. Pryor, S.A. Cuccaro, M. Mascagni, M.L. Robinson, Implementation and usage of a portable and reproducible parallel pseudorandom number generator, In: Proceedings of Supercomputing '94, IEEE, New York, 1994, pp. 311–319.
[29] J. Saarinen, K. Kankaala, T. Ala-Nissila, 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.
[30] F. Schmid, N.B. Wilding, Errors in Monte Carlo simulations using shift register random number generators, Int. J. Mod. Phys. C 6 (6) (1995) 781–787.
[31] W. Selke, A.L. Talapov, L.N. Schur, Cluster-flipping Monte Carlo algorithm and correlations in "good" random number generators, JETP Lett. 58 (8) (1993) 665–668.
[32] SPRNG—scalable parallel random number generators. Available at: SPRNG 1.0—http://www.ncsa.uiuc.edu/Apps/SPRNG; SPRNG 2.0—http://sprng.cs.fsu.edu.
[33] A. Srinivasan, D.M. Ceperley, M. Mascagni, Random number generators for parallel applications, In: D.M. Ferguson, J.I. Siepmann, D.G. Truhlar (Eds.), Monte Carlo Methods in Chemical Physics, in: Advances in Chemical Physics, vol. 105, John Wiley and Sons, New York, 1999, pp. 13–36.
[34] I. Vattulainen, Framework for testing random numbers in parallel calculations, Phys. Rev. E 59 (1999) 7200–7204.
[35] I. Vattulainen, T. Ala-Nissila, K. Kankaala, Physical tests for random numbers in simulations, Phys. Rev. Lett. 73 (1994) 2513–2516.