Testing Parallel Random Number Generators

Ashok Srinivasan (ashok@math.iitb.ernet.in)[1,2], David Ceperley (ceperley@ncsa.uiuc.edu)[2], and Michael Mascagni (Michael.Mascagni@usm.edu)[3]

[1] Department of Mathematics, Indian Institute of Technology–Bombay, Powai, Mumbai 400 076 **India**
[2] National Center for Supercomputing Applications, 405 N. Matthews Av., Urbana, IL 61801 **USA**
[3] Department of Mathematics, Box 10057, University of Southern Mississippi, Hattiesburg, MS 39406-0057 **USA**

Abstract. A parallel random number generator (PRNG) must be tested for two types of correlations – (i) Intra-stream correlation, as for any serial 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 empirical tests are necessary. Many of the popular RNGs in use today were tested when computational power was much lower, and hence they were evaluated with much smaller.

This paper describes several tests of PRNGs, both statistical and physically-based tests. We show defects in several popular generators. We then present the results for the tests conducted on the SPRNG generators. These generators have passed some of the largest empirical random number tests ever undertaken.

1 Introduction

Monte Carlo (MC) computations have, currently do, and will continue to consume a large fraction of all available high-performance computing cycles. MC methods can be defined as any numerical process that consumes random numbers. Thus the results of MC computations can be adversely affected by defects in the random number sequence used. Many of the random number generators (RNGs) in use today were developed and tested when the speed of the computers was much less than today. With increase in the speed of computers, many more random numbers are now consumed in even ordinary MC computations. This has revealed defects in the RNGs that were not apparent with smaller simulations [1–4]. 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 [5,6,1,7–9] both theoretically and empirically.

One of the most common methods of MC parallelization is to use the same algorithm on each process, except with a different random number stream. 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 that non-combined results. There is, then, a need to check for two types of defects in the PRNGs: (i) One must verify that each random number stream on each process appears to be random. That is, there should be no apparent correlation between the elements of a single random number sequence. We refer to this as the absence of intra-stream correlation. (ii) There should be no correlation between streams on the different processors. We call this the absence of inter-stream correlation.

We shall next describe popular methods of designing parallel random number generators (PRNGs). In the methods of cycle division, a cycle corresponding to a single random number generator is partitioned among the different processors so that each processor gets a different portion of the cycle. This partitioning is generally performed in the following three ways: (i) naive users randomly select a different seed on each process and hope that the seeds will take them to widely separated portions of the cycle, so that there will be no overlap between the portions of the sequence used by the different processors. (ii) in the *cycle splitting* scheme, the user deterministically chooses widely separated seeds for each process. The danger here is that if the user happens to consume more random number than expected, then the sequences could overlap. Apart from this, generators often have long range correlations [10,11]. These long range correlations in the sequential generator become short range inter-stream correlation in the parallel generator. (iii) 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}.... This again has problems because long range correlations in the original sequence can become short-range *intra-stream* correlations. Apart from these defects, the method of sequence splitting 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.

The parallelization scheme we recommend is based on *parameterization*. We can parameterize a set of streams by a *stream number*. Given the stream number i, there is an easy way of producing the i th stream. These parameterizations are done in two ways: (i) *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 process a seed from a different cycle. (ii) *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. For this we need a way of parameterizing the iteration function so that given i, the i th iteration function can be easily produced.

We have developed the `SPRNG` [12] software package that has a set of well tested PRNGs based on parameterization. They have been subjected to the largest empirical RNG tests of which we are aware. The package also comes with a suite of popular sequential and parallel tests to enable others to test their own generators.

The outline for the rest of the paper is as follows. In section (2) we describe the general types of sequential and parallel random number tests we have performed, and explain a few in greater detail. In section (3) we give the results of our tests. Finally, in section (4) we give our recommendations on the use of the different generators based on these tests. The appendix describes the generators implemented in the SPRNG package.

2 Description of Tests

A good parallel random number generator 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 1-dimension, but in higher dimensions as well. In fact, many simulations require the absence of correlations for a large number (say thousands) of dimensions.

RNG tests, both sequential and parallel, can be broadly classified into two categories: (i) statistical tests and (ii) physically-based tests. The basic idea behind the statistical tests is that the random number streams obtained from a generator should have the properties of an i.i.d. random sample drawn from the uniform distribution. Tests are designed so that the expected value of some test statistic is known for the uniform distribution. The empirically generated random number stream is then subjected to the same test, and the statistic obtained is compared against the expected value. While a boundless number of tests can be constructed, certain tests have become popular and are accepted as de facto standards: (i) the series of tests proposed by Knuth [13] and (ii) the DIEHARD tests implemented by Marsaglia [14]. Generators that pass these tests are considered "good".

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

2.1 Statistical Tests

We now describe in detail a small number of tests from those proposed by Knuth.

Collisions test: We concatenate the k most significant bits from d random integers to form a new $\log_2 m = kd$ bit random integer. We form $n << m$ 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 d-dimensional correlation. It is one of the most effective tests among those proposed by Knuth.

Gap test: We generate floating point numbers in [0,1] and note the gap in the sequence between successive appearances of numbers in a fixed 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 several such gaps and compare the distribution of these gaps with the expected one.

Parallel Statistical Tests: The tests described above are for a sequential generator. We can modify them by interleaving different streams to produce a new random number stream which is then subjected to the same sequential tests mentioned above. 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.

There are certain other tests that are inherently parallel, in contrast to the above scheme which is really a parallelization of a sequential test. We describe two tests below.

Blocking test: We can use the fact that the sum of independent variables asymptotically approaches the normal distribution to test for the independence of random number streams. We add random numbers from several streams and form a sum. We generate several such sums and check if their distribution is normal.

Fourier Transform test: We fill a two dimensional array with random numbers. Each row of the array is filled with random numbers from a different stream. We calculate the Fourier coefficients and compare with the expected values. This tests is repeated several times and we check if there are particular coefficients that are repeatedly "bad".

2.2 Physically-based Tests

Physically-based tests use random numbers in a manner similar to they way they are used in practical applications. In some cases, the exact behavior of the test may be known analytically. For statistical mechanical applications, the two-dimensional Ising model (a simple lattice spin model) is often used since the exact answer is known. Since the Ising model is also known to have a phase transition, this system is sensitive to long-range correlations in the PRNG. There are several different algorithms, such as the Metropolis and the Wolff algorithms, that can be used to simulate the Ising model, and the random numbers enter quite differently in each algorithm. Thus this application is very popular in testing random number generators and has often detected defects in generators [8,15]. We can test parallel generators on the Ising model application by assigning distinct random number sequences to different subsets of lattice sites [15]. In our parallel tests we use a different stream for each site.

3 Test Results

We next describe the results of our tests. We first show that the popular 32 bit Unix LCG, *rand* fails even sequential RNG tests. 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 with different seeds. Thus this generator fails with about 10^7 to 10^8 random numbers. On a machine capable of generating 10^7 random numbers per second, this would take between 1 and 10 seconds! Therefore this generator should not be used any longer.

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} random numbers. The blocking test was performed with 128 streams. 256 random numbers from each stream were added together to form a sum, and 10^6 such sums were generated. The sequence splitting was performed by splitting the sequence 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 order to give the parallelization a fair chance of demonstrating its utility, we did not use such a worst-case situation. The generator still failed this test, demonstrating the danger of cycle division.

With parallelization through parameterization, one still needs to be careful about the seeding. If all the streams start with the same initial state, then the generator fails the Metropolis algorithm as shown in Fig. 1, though the sequential tests are passed (implying that each stream is sufficiently random). The blocking test too is failed by this generator with such a seeding scheme, for around 10^{10} random numbers, demonstrating the effectiveness of this test. When the seeds are staggered, so that each stream starts at a sufficiently different position in the sequence, the tests are passed, as shown by the solid line of Fig. 1. Thus one needs to be careful about seeding, especially because these correlations between the streams persist even when we discard the first few million random numbers before starting the tests.

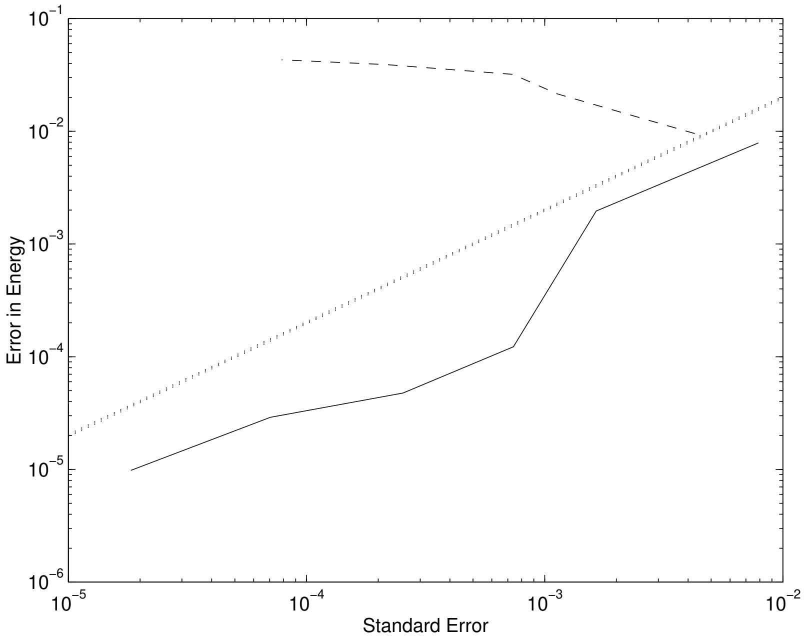

Fig. 1. Plot of the actual error versus the internally estimated standard deviation of the energy error for Ising model simulations with the Metropolis algorithm on a 16×16 lattice with a different Linear Congruential sequence at each lattice site. The dashed line shows the results when all the Linear Congruential 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.

The popular Unix additive lagged-Fibonacci generator (LFG), *random*, with a lag of 31, fails even the serial gap test with the following parameters: $[a, b] = [0.5, 0.51]$, gaps of length greater than 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} random numbers. The Birthday Spacings test in the DIEHARD package is more effective and detects the defects in additive LFGs with much fewer random numbers. This generator also fails serial physically-based tests. For example, the Wolff algorithm fails with around 10^8 random numbers for a 16×16 lattice, as shown in Fig. 2. The `SPRNG` LFG combines two different random number streams to produce a new stream. This generator passes the gap test even with 10^{13} random numbers used, which we believe is the largest ever test of random numbers. It also passes the physically-based tests, both serial and parallel, with around 10^{11} random numbers.

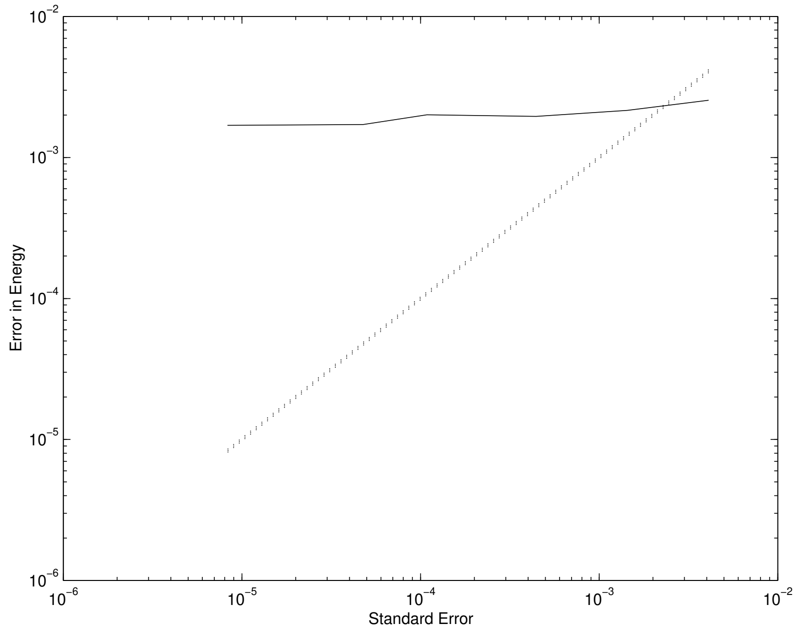

Fig. 2. Plot of the actual error versus the internally estimated standard deviation of the energy error for Ising model simulations with the Wolff algorithm on a 16×16 lattice with *random()* additive lagged Fibonacci generator (solid line). The dotted line is an error equal to the standard deviation.

Among the tests described in Knuth, the Collisions tests is one of the most effective. All the `SPRNG` generators pass these tests with 10^{12} random numbers per test. Some of the parameters for this case were: $d = 4$, $k = 5$ for 4-dimensional correlation, and $d = 10$, $k = 3$ for 10-dimensional correlation. 1024 streams were tested. 1000 and $50,000$ tests were done on each stream in the serial and parallel tests respectively.

We shall now give a summary of the tests results. All the `SPRNG` generators were tested with the DIEHARD suite, and they all passed these tests, except that the lower order bits of the 48 bit LCG are bad. The following tests from Knuth [13] were performed, including their parallel versions: collisions, coupon collector, equidistribution, gap, maximum of t, permutations, poker, runs-up, serial. At least $O(10^{11})$ random numbers were tested for each generator in each case. The collisions test used 10^{12} random numbers. The serial gap test for the Additive LFG used 10^{13} random numbers – the largest empirical random number test ever accomplished. The parallel gap test for this generator used 10^{12} random numbers. We also performed the blocking test for parallel generators, and the Metropolis and Wolff algorithm for the Ising model with at least 10^{11} random numbers. More details can be found at our web site.

4 Conclusions

In this section we shall describe the weaknesses of different generators and give general guidelines on their use.

LCGs with power-of-two modulii are known to have extremely nonrandom lower order bits – the i th LSB has 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 serial. The 48 bit generator is much better, and of course, the 64 bit generator is still better. Despite this problem, the 48 bit 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 CMRG. This improves the quality of the 32 MSBs. However, the lower order bits are still identical to those of the corresponding LCG.

The prime modulus LCG 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 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 lagged-Fibonacci generator (LFG) implemented in the SPRNG package actually combines two different random number 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 (2). To be on the safe side, we suggest using a large lag. The multiplicative LFG is the only generator considered here that has a non-linear recurrence relation. We expect it to be safe to use it even with a small lag.

We conclude this paper by recommending the technique of independent streams for parallelizing random number generators. Using the technique of cycle division can turn long range correlations in the original sequence into short range inter- or intra-stream correlation. There have been some warnings about the dangers of using independent streams [16] 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.

Acknowledgement

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.

A SPRNG Generators

We next describe the generators implemented in SPRNG . These generators have passed the tests described above, and are among the generators that have been the best tested empirically.

1. **Combined Multiple-Recursive Generator:** `cmrg`
   This generator is defined by the following recursion:

   $$z_n = x_n + y_n \times 2^{32} (mod 2^{64})$$

   where x_n is the sequence generated by the 64 bit Linear Congruential Generator given below and y_n is the sequence generated by the following prime modulus Multiple Recursive Generator:

   $$y_n = 107374182 \times y_{n-1} + 104480 \times y_{n-5} (mod 2147483647)$$

   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}. The number of distinct streams available is over 10^8.

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:

$$x_n = a \times x_{n-1} + p(mod 2^{48})$$

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. The period of this generator is 2^{48}. The number of distinct streams available is of the order of 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}. The number of distinct streams available is over 10^8.

4. **Modified Lagged-Fibonacci Generator:** `lfg`
The recurrence relation for this sequence of random numbers is given by the following equation:

$$z_n = x_n XOR y_n$$

where XOR is the exclusive-or operator, x and y are sequences obtained from additive LFG sequences X and Y of the following form:

$$X_n = X_{n-k} + X_{n-l}(mod 2^{32})$$

$$Y_n = Y_{n-k} + Y_{n-l}(mod 2^{32})$$

l and k are called the lags of the generator, and we use the convention that $l > k$. 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 Lagged Fibonacci Generator is performed in order to avoid certain correlations that are observed in the unmodified generator.
The period of this generator is $2^{31}(2^l - 1)$ where l is the lag. For the default generator with lag $l = 1279$, the period is approximately 2^{1310}. The number of distinct streams available is $2^{[31(l-1)-1]}$. For the default generator this gives $2^3 9648$ distinct streams.
The sequence obtained is determined by the l initial values of the sequences X and Y. The seeding algorithms ensures that distinct streams are obtained during parallelization.

5. **Multiplicative Lagged-Fibonacci Generator:** `mlfg`
The recurrence relation for this sequence of random numbers is given by the following equation:

$$x_n = x_{n-k} \times x_{n-l}(mod 2^{64})$$

The period of this generator is $2^{61}(2l-1)$ where l is the larger lag. For the default generator with lag $l=17$, the period is approximately 2^{81}. The number of distinct streams available is $2^{[63(l-1)-1]}$. For the default generator this gives around 2^{1008} distinct streams.
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.

6. **Prime Modulus Linear Congruential Generator:** `pmlcg`
   This generator is defined by the following relation:

$$x_n = ax_{n-1}(mod 2^{61} - 1)$$

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

References

1. 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.
2. P. Grassberger. On correlations in 'good' random number generators. *Phys. Lett. A*, 181(1):43–46, 1993.
3. 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.
4. 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.
5. M. Mascagni S. A. Cuccaro 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. 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.
7. 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.
8. P. Coddington. Analysis of random number generators using Monte Carlo simulation. *Int. J. of Mod. Phys. C*, 5(3):547–560, 1994.
9. P. L'Ecuyer, J.-F. Cordeau, and R. Simard. Close-point spatial tests for random number generators. *Operations Research (To appear): http://www.iro.umontreal.ca/ lecuyer/papers.html (npair.ps)*, 1999.

10. A. De Matteis and S. Pagnutti. Parallelization of random number generators and long-range correlations. *Parallel Comput.*, 15:155–164, 1990.
11. A. De Matteis and S. Pagnutti. Long-range correlations in linear and non-linear random number generators. *Parallel Comput.*, 14:207–210, 1990.
12. SPRNG – Scalable Parallel Random Number Generators. http://www.ncsa.uiuc.edu/Apps/SPRNG.
13. D. E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Second edition*. Addison-Wesley, Reading, Massachusetts, 1981.
14. G. Marsaglia. Diehard. ftp://stat.fsu.edu/pub/diehard.
15. P. Coddington. Tests of random number generators using Ising model simulations. *Int. J. of Mod. Phys. "C"*, 7(3):295–303, 1996.
16. P. Coddington. Random number generators for parallel computers, 28 April 1997. www.npac.syr.edu/users/paulc/papers/NHSEreview1.1/PRNGreview.ps.