DHPC-025

Techniques for Empirical Testing of Parallel Random Number Generators

Paul D. Coddington
Department of Computer Science, University of Adelaide
Adelaide, SA 5005, Australia
paulc@cs.adelaide.edu.au

Sung-Hoon Ko
Northeast Parallel Architectures Center, Syracuse University
111 College Place, Syracuse, NY 13244, U.S.A.
shko@npac.syr.edu

16 January 1998

Abstract

Parallel computers are now commonly used for computational science and engineering, and many applications in these areas use random number generators. For some applications, such as large-scale Monte Carlo simulations, it is crucial that the random number generator have good randomness properties. Many programs are available for testing the quality of sequential random number generators, but very little work has been done on testing parallel random number generators. We present some techniques for empirical testing of random number generators on parallel computers, using tests based on computational science applications as examples. In particular, we focus on tests based on parallel algorithms developed for Monte Carlo simulations of the two dimensional Ising model, for which exact results are known. Preliminary results of these tests are presented for several parallel random number generators.

1 Introduction

Parallel computers are now commonly used for computational science and engineering. Many of these applications use parallel implementations of random number generators. Since random numbers are in practice computed using deterministic algorithms, these are more accurately called pseudo-random number generators. In some applications, the quality of the pseudo-random numbers (i.e. how closely they resemble truly random sequences) is not that important. However in many of the applications for which random number generators are most heavily used, such as Monte Carlo simulations [10], the quality of the random number generator is crucial, since an inadequate random number generator can produce incorrect results. This is especially true in large-scale simulations on supercomputers, which consume huge quantities of random numbers, and for which vector or parallel algorithms for random number generation are required.

As noted in several review articles [24, 21, 25, 31, 3], sequential random number generators provided by computer vendors or recommended in computer science texts have often been (and unfortunately continue to be) of poor quality. Even generators that perform well in standard statistical tests for randomness may be unreliable for particular applications, as has been seen many times in the computational science literature, particularly for large-scale Monte Carlo simulations [22, 30, 16, 27, 15, 40]. This has led to the development of a number of "physical" tests based on standard computational science applications, such as Monte Carlo simulation or random walks [15, 13, 38].

The many problems caused in the past by inadequate sequential random number generators are likely to be repeated in a new generation of simulations using parallel computers, unless users are provided with parallel random number generators that have been carefully studied and tested. Few rigorous mathematical results are known about the randomness properties of random number generators, especially for parallel algorithms, so a generator should be subjected to stringent and varied empirical tests before being used.

There has been quite a lot of research on developing algorithms for vector and parallel random number generators (see the reviews by Anderson [5], Brent [11] and Coddington [3]), but very little work has been done on developing and applying methods for testing such generators.

New approaches are necessary to test algorithms for generating random numbers on parallel computers, for example to look for correlations between random number streams on different processors, or to parallelize existing sequential tests, particularly the empirical tests that have been so successful at weeding out inadequate generators.

Here we give an overview of methods for empirical testing of parallel random number generators, concentrating on parallel implementations of physical tests such as Monte Carlo simulations. We also present some examples of results of these tests for several different kinds of parallel random number generator. These tests are still being done on many different parallel generators, and more comprehensive results will be presented elsewhere [46].

2 Parallel Random Number Generators

Random number generators use iterative deterministic algorithms for producing a sequence X_i of pseudo-random numbers that approximate a truly random sequence. The main algorithms used for sequential random number generators are:

Linear congruential generators (LCGs)

$$X_i = A * X_{i-1} + B \mod M$$

which we denote by $\mathrm{L}(A, B, M)$.

Lagged Fibonacci generators (LFGs)

$$X_i = X_{i-P} \odot X_{i-Q}$$

which we denote by $\mathrm{F}(P, Q, \odot)$, $P > Q$, where $\odot$ is any binary arithmetic operation, such as addition or multiplication modulo M, or the bitwise exclusive OR function XOR.

Shift register generators can usually be defined in terms of LFGs using XOR, however these are of lower quality than equivalent LFGs using addition or multiplication.

Combined generators that combine (usually by addition modulo M) the results of two or more generators, usually two LCGs, two LFGs, or an LCG plus an LFG.

For more information, see one of the many books or review articles on random number generators [24, 21, 25, 31, 3].

The main techniques used for parallelizing random number generators involve distributing the sequences of random numbers produced by a sequential generator among the processors (or abstract processors for data parallel languages) in the following different ways [5, 11, 3]:

Leapfrog – the sequence is partitioned among the P processors in a cyclic fashion, like a deck of cards dealt to card players;

Sequence splitting – the sequence is partitioned among processors in a block fashion, by splitting it into non-overlapping contiguous sections;

Independent sequences – the initial seeds are chosen in such a way as to produce long period disjoint subsequences on each processor.

There are other methods for implementing parallel random number generators, including using a different generator (or the same type of generator but with different parameters) for each processor, but the above methods are the most commonly used.

An obvious requirement for a good parallel random number generator is that the sequential generator on which it is based should have acceptable randomness properties. Unfortunately, many of the widely-used parallel generators fail even this first requirement. Recent physical tests have shown that many generators in common use, particularly shift register generators and lagged Fibonacci generators with a small lag, are inadequate for many applications [15, 13, 38, 3].

Even when a good sequential generator is used, it is not guaranteed that it will produce a good parallel generator [3]. Some generators have periods that may be adequate for current sequential workstations, but not for Teraflop supercomputers. Parallelization, particularly using leapfrog or sequence splitting, may amplify small correlations in the sequential generator. Also, great care needs to be taken in initializing the generators, so that the seeds are not correlated over different processors. It is therefore prudent to subject any parallel random number generator to a battery of empirical tests.

3 Testing Parallel Random Number Generators

Over the years many widely-used methods for generating pseudo-random numbers have been shown to be inadequate, either by theoretical arguments, or empirical tests, or both. In some cases theoretical arguments can show that there are correlations in the sequence of numbers, however in many cases the problems only show up in empirical tests that statistically compare the results produced by the random number generators with results expected from a truly random sequence of numbers. Many standard statistical tests of this kind are available for sequential random number generators [24, 26, 47].

Some of these sequential tests have been applied to parallel generators, by testing the random number streams on each processor, or the combined stream from all processors [12, 5, 1]. This is the usual approach in testing parallel generators. However, very little work has been done on developing tests specifically for parallel random number generators, particularly for physical tests.

3.1 Physical Tests

In addition to standard statistical tests, it is useful to apply physical tests that are related to the various computational science applications for which random numbers are commonly used. As with the statistical tests, these tests generally compare results obtained from using a pseudo-random number generator with known exact results that would occur if the numbers generated were truly random.

Tests of this kind include simulations of exactly solvable systems such as the two dimensional Ising model [23, 20, 15, 13, 38], percolation models [39], and random walks [40, 39, 38]. Vattulainen *et al.* [38] have developed a software package for testing sequential random number generators using a variety of physical tests, including Ising model simulations and random walks. Generators that pass standard statistical tests have often been found to fail these physical tests. It is therefore important to use as wide a variety of these empirical tests as possible.

It should be noted that any application can be used to test random number generators, by simply comparing results obtained with two different generators. Since no amount of testing can ever determine whether a random number generator will work for a particular application, it is always a good idea to run an application program using more than one generator, in order to check the results.

3.2 Ising Model Monte Carlo Tests

We have developed both sequential and parallel programs for testing random number generators using Monte Carlo simulation of the two dimensional Ising model [13, 48]. This simple model has been solved exactly for a finite lattice of grid points [17], so that values of the energy and the specific heat (the variance of the energy) of the system calculated from the Monte Carlo simulation can be compared with the known exact values. An added advantage of this approach is that there are several different Monte Carlo algorithms that can be used to simulate the Ising model, and each of them uses random numbers in a different way.

We have implemented the three most widely used methods: the Metropolis algorithm [10] which updates a single site of the lattice at a time; the Swendsen-Wang (SW) algorithm [33] which forms clusters of sites to be updated collectively; and the Wolff algorithm [41] which updates a single cluster of sites. The Swendsen-Wang and Wolff cluster update algorithms are extremely efficient and allow very precise Monte Carlo simulations of the Ising model, easily reducing statistical errors to better than one part in 10^4 on small lattices. This precision provides us with a very effective practical test of the randomness of a pseudo-random number generator, and in particular its suitability for Monte Carlo simulation.

We have implemented the parallel Monte Carlo Ising model programs using both message passing and data parallel languages. The message passing programs were written in C and Fortran, with versions using Express [28, 29], CMMD [43], or MPI [44]. The data parallel programs were written in CM Fortran [35], MP Fortran [36], or High Performance Fortran (HPF) [4].

The Metropolis algorithm is regular and local and can therefore be easily and efficiently parallelized using message passing or data parallel languages. A standard block domain decomposition and red/black updating scheme is used [18, 19].

The main computational task in the Swendsen-Wang algorithm is identifying and labeling the clusters of connected sites, which is equivalent to the problem of connected component labeling of an undirected graph [49]. Since the Swendsen-Wang clusters may be highly irregular (fractal, in fact) and highly non-local (they can span the lattice), this is a difficult problem to parallelize efficiently. The message passing version of the Swendsen-Wang program uses the local label propagation or self-labeling algorithm [14, 8, 34] to do the component labeling, which is reasonably efficient on MIMD machines, as long as the lattice size is fairly large. This program has been used for high precision Monte Carlo studies of Ising and Potts spin models [9]. The data parallel program uses a different algorithm to label the clusters [6], and as expected for this kind of irregular, non-local problem, it is much less efficient.

We have also developed message passing and data parallel implementations of the Wolff algorithm [7]. In this case the main computational task is to compute the expansion of the edge of the single cluster, which can be done fairly efficiently if a cyclic data distribution is used. Again, the message passing implementation is much more efficient than the data parallel implementation. For this reason, we have not done any testing using the data parallel Wolff algorithm on SIMD machines. However, we are planning to implement an HPF version of this program, which will allow us to run the test on a single processor (since HPF programs give the same result on any number of processors), which should be more efficient.

3.3 Using Replicated Sequential Test Programs

The usual approach to testing parallel random number generators has been to use standard sequential tests on the random number streams on each processor, or the combined stream from all processors [12, 5, 1].

The simplest type of empirical test is to use the pseudo-random number generator with the test program to compute a result and a statistical error for that result, and then compare it with the known exact value which would occur for a truly random sequence. For physical tests such as Monte Carlo, the result is usually a mean value (such as the energy for the Ising model), and the error in the mean is easily computed using standard techniques, such as binning the measurements, that take into account possible correlations in the measurements [10, 13]. It is also useful to measure the variance of the result (for the Ising model, this is the specific heat), since in some cases the correlations in the generator may be such that the mean is correct, but the variance is not, particularly for parallel generators [2].

To check the quality of the random number generator, we simply compute the deviation $\Delta = (\overline{x} - \langle x \rangle)/\sigma$ between the computed (sample) mean value $\overline{x}$ and the known exact value (or expectation value) $\langle x \rangle$, as a multiple of the error in the mean σ. In the usual way, we can figure out the probability of obtaining a particular value of the deviation, for example, $\Delta > 3.3$ should occur with probability 0.001 [45], so we should be suspicious of a generator that produces such a large deviation from the exact value.

A better test is to do multiple independent runs of the test program, each of which uses different initial values to seed the random number generator. This allows some checking that the quality of the result is independent of the seed values, and that subsequences produced by different seeds are uncorrelated. In this case, we can test the deviation from the exact result for each run, as well as for the average over all runs. We can treat the results for each of these N runs as independent data points, so it is very easy to compute an error in the mean for the N combined results (as long as N is large enough to give a reliable error estimate).

In addition, this method provides an additional check for possible discrepancies in the statistical fluctuations expected between the results for each run, by computing the chi-squared per degree of freedom χ^2 for the N data points [45]. If the chi-squared value is too large, then on average the values are too far away from the expected result, whereas if the chi-squared is too small, then the results are correlated in some way. $\chi^2 > 2.0$ or $\chi^2 < 0.34$ should occur with probability less than 0.001 for a truly random generator [45].

In previous work on physical tests of sequential random number generators [13], we used this approach with $N = 25$ independent runs. In some cases, the generators passed the simple test of the deviation from the expected value, but failed the chi-squared test.

This kind of chi-squared test can also be used to check for correlations between the random number streams on different processors for a parallel random number generator, by replicating the sequential test across multiple processors. This involves a simple parallelization of the sequential test program, so that an independent run with a separate I/O stream is performed on each processor, and a parallel random number generator is used.

For parallel generators that use independent parallelism, the only difference between this parallel test and the use of multiple copies of the sequential generator in the sequential tests is in how the generators on each processor are initialized, however the initialization is crucial for generators of this kind, so this is still a useful test.

We have implemented these replicated sequential tests using the three different Ising model algorithms described in section 3.1. We are also working on a similar implementation of the physical tests of Vattulainen *et al.* [38]. These tests should be particularly useful for studying the effects of different initializations of parallel generators. This work is still in the very early stages, so we cannot present useful results as yet.

3.4 Using Parallel Test Programs

Another approach to testing parallel random number generators is to use parallel versions of the test programs. This may be more effective at finding subtle correlations between random number streams on different processors.

In this case the tests are run and the results analyzed in the same way that the sequential version of the test is run for a sequential generator on a sequential computer. For a physical test such as Monte Carlo simulation of the Ising model, N independent tests are run on a parallel computer (we have chosen $N = 25$), with different initial seeds for each test, and the values of the deviation Δ and the chi-squared per degree of freedom χ^2 are computed.

For data parallel generators and test programs (e.g. in HPF), the results are independent of the number of physical processors used, so each test could actually be run on a single processor, which would be more efficient. This may not be the case for message passing (MIMD-style) implementations, so the number of processors used may be an additional test parameter, as well as the problem size (or the number of abstract processors).

As outlined in Section 3.1, we have implemented parallel versions of Ising model tests using three different Monte Carlo algorithms. It would also be possible to parallelize other physical tests, such as percolation models.

4 Some Results

We have tested several parallel random number generators using and fully parallel test programs described in section 3.3. This work is still in progress and only preliminary results are given here. Other parallel random number generators will be tested, and more comprehensive results will be given elsewhere [46].

The parallel random number generators were tested using a variety of parallel computers, including Thinking Machines CM-2 and CM-5, Maspar MP-100, Intel iPSC/860, nCUBE/2, IBM SP-2, and DEC Alpha and Sun workstation clusters. The message passing programs were run on 16 processors, except for runs on a 32-processor CM-5. The results of the data parallel programs are dependent on the number of abstract processors, or data elements (i.e. the lattice size for this application), rather than the number of physical processors used.

The following parallel random number generators have been tested:

1. `CMF_RANDOM`, a parallel cellular automata generator used on the Connection Machine [35, 42].

2. `CMSSL FAST_RNG`, the lagged Fibonacci generator $F(17,5,+)$ used in the Connection Machine Scientific Software Library (CMSSL), with the lag recommended in the CMSSL user guide [37], initialized using the `CMF_RANDOM` generator.

3. `CMSSL VP_RNG`, which is the same as `FAST_RNG`, but the LFG is replicated over each virtual (or abstract) processor, rather than each physical processor.

4. `P_RANDOM`, a parallel version of the standard Unix and C lagged Fibonacci generator `random`, replicated over abstract processors. We tested both the original (`P_RANDOM #1`) and the more recent (`P_RANDOM #2`) versions implementated by Maspar [36], which differ in how the seeds are initialized.

5. `PRAND`, the standard 32-bit C and Unix linear congruential generator `RAND`, $L(1103515245,12345,2^{31}-1)$, parallelized over physical processors using a leapfrog technique [18].

6. `IBM XLHPF 48-bit`, the 48-bit linear congruential generator $L(44485709377909,0,2^{48})$ parallelized over abstract processors using a leapfrog technique.

We have used the same techniques for generating and analyzing the data as were used in previous work on Monte Carlo testing of sequential random number generators [13]. For each random number generator, 25 independent simulation runs with different initial seeds were performed for each different test (Metropolis, SW, and Wolff). Each simulation was at least 10^5 Monte Carlo sweeps of a 128×128 lattice at the critical point of the 2-d Ising model. The quantity of random numbers generated for all 25 simulation runs was of order 10^{11} in total for each different test. In some cases we have also tested the generators using alternate lattice sizes (64×64 and 256×256), which can probe for correlations at different scales.

The results for the parallel tests are shown in Table 1 for the data parallel (SIMD) results and Table 2 for the message passing (MIMD) results.

One point to note from the results is that good initialization (or seeding) of parallel random number generators is crucial to their performance, particularly for parallel lagged Fibonacci generators, where many seeds need to be assigned on each processor. The original version of `P_RANDOM` for the Maspar had very naive initialization, and the generator was extremely poor. A change to the initialization routines greatly improved the performance, but still not enough for it to pass all the tests.

In some cases, such as `CMF_RANDOM`, the generator only fails the test for certain lattice sizes, passing the Metropolis test for 128^2 but failing for 64^2 and 256^2. Note that the CMSSL `FAST_RNG` routine has the same behaviour, presumably because it is initialized using `CMF_RANDOM`.

We might have expected that `PRAND` would fail the tests, since the period of this generator is less than the number of random numbers used in the test. However it only fails for the Metropolis test, which seems to be better at picking up correlations in linear congruential generators. A more interesting result is that the 48-bit LCGs also fail this test. This is probably because the modulus used (2^{48}) is a power of two, which is known to give correlations in the sequence at intervals that are powers of two [32]. This is particularly bad for parallelization using the leapfrog method, since the number of processors (or abstract processors) is often a power of two (as it is in the Metropolis test), which tends to amplify these correlations [22, 16, 3].

The Metropolis test appears to be tougher than the Wolff or SW tests for all parallel generators, which is not the case for sequential generators [13].

		Energy		Specific Heat	
Generator	Lattice	SW	Metrop	SW	Metrop
`CMF_RANDOM`	64×64	-0.96	**-10.68**	0.60	**5.07**
		0.83	**8.75**	0.74	**2.72**
	128×128	1.03	-0.24	0.42	-1.61
		1.20	0.92	1.11	1.49
	256×256	–	**-7.13**	–	1.23
		–	**4.93**	–	**2.41**
`CMSSL VP_RNG`	64×64	1.28	0.02	0.02	1.24
		1.83	1.02	1.65	1.00
	128×128	-0.17	0.72	2.57	-2.08
		1.37	1.24	1.25	1.83
`P_RANDOM #1`	128×128	–	**53.04**	–	**-10.34**
		–	**201.83**	–	**17.24**
`P_RANDOM #2`	128×128	–	1.90	–	-2.55
		–	**3.14**	–	**4.44**
	256×256	–	-0.20	–	-1.64
		–	**4.47**	–	**7.74**
`IBM XLHPF`	128×128	–	1.00	–	-0.42
		–	**2.45**	–	1.84

Table 1: Results of Monte Carlo simulations of the 2-d Ising model using different data parallel random number generators on SIMD parallel computers. The first line for each generator shows the deviation of the Monte Carlo results from the exact values, as a multiple of the error in the mean. The second line shows the χ^2 per degree of freedom. A dash means the test has not been done. Numbers in bold type indicate results which would occur with a statistical probability of less than 0.001 for true random sequences.

5 Conclusions

Since faster computers and better algorithms are rapidly improving the precision of Monte Carlo and other stochastic simulations in computational science, it is important to continue to search for better parallel random number generators with very long periods, and in particular to make more precise and varied tests of the randomness properties of these generators.

Although a lot of research has been done on developing improved parallel random number generators, little has been done on developing stringent empirical tests for such generators, particularly physical tests based on computational science applications. These have proven to be very useful in identifying problems in sequential random number generators, and parallel versions of physical tests are likely to be equally useful in testing parallel random

		Energy			Specific Heat		
Generator	Lattice	SW	Wolff	Metrop	SW	Wolff	Metrop
`CMSSL FAST_RNG`	64×64	0.06	-0.99	**-7.33**	0.79	0.34	1.93
		-0.30	1.27	**5.82**	1.19	1.64	1.16
	128×128	-0.21	0.12	-1.97	0.99	-1.01	-0.30
		0.17	0.78	1.00	1.23	1.16	0.52
`F(1279,1063,+)`	64×64	?.??	0.09	?.??	?.??	-0.63	?.??
		?.??	0.93	?.??	?.??	1.22	?.??
	128×128	1.02	-0.35	0.96	1.05	0.60	0.63
		1.48	0.93	**2.35**	0.69	1.24	1.27
`PRAND`	64×64	-0.27	0.66	**6.97**	1.13	-0.23	**-4.88**
		1.07	1.26	**5.28**	1.09	1.57	**5.17**
	128×128	-1.88	–	**221.6**	-0.07	–	**-55.21**
		0.79	–	**10350**	0.85	–	**629.6**

Table 2: As for Table 1, but for message passing implementations of parallel random number generators on MIMD parallel computers.

number generators.

We have described two different types of tests for parallel random number generators – replicated sequential tests and fully parallel tests. We have created parallel implementations using both these approaches for three physical tests, which are based on three different Monte Carlo algorithms for simulating the 2D Ising model. The fully parallel tests have been applied to several commonly-used parallel random number generators. Many of the generators failed these tests, indicating that thorough testing of generators is very important, and that physical tests can be very powerful in this regard.

The quality of a parallel random number generator can be heavily dependent on the initialization of the generator. We plan to study this further in the future, particularly using the replicated sequential tests.

One lesson from these results is not to trust random number generators provided by computer vendors. In the past, many inadequate generators have been provided or recommended for sequential computers [24, 31], and a similar problem is now occurring with generators for parallel and vector machines. This may cause problems for parallel applications in computational science and engineering, particularly large-scale Monte Carlo simulations.

Problems in random number generators may not show up in empirical tests until large quantities of random numbers are used. Statistical tests of commonly-used generators that were done many years ago on outdated sequential computers using perhaps a few million random numbers are likely to be irrelevant when a parallel version of the generator is used for large-scale Monte Carlo simulations on modern-day supercomputers, which may use more than 10^{12} random numbers. Empirical tests should be periodically repeated using faster computers and larger quantities of random numbers, which better reflect the conditions to which they are subjected in current computational science applications.

Acknowledgements

This work was supported in part by the the Center for Research on Parallel Computation with NSF cooperative agreement No. CCR-9120008, and by ARPA under contract No. DABT63-91-K-0005, and also by the Distributed High Performance Computing Infrastructure (DHPC-I) project of the Research Data Networks (RDN) Cooperative Research Center

(CRC), managed under the On-Line Data Archives (OLDA) Program of the Advanced Computational Systems (ACSys) CRC. RDN and ACSys are established under the Australian Government's CRC Program.

The tests were run using computers at the Northeast Parallel Architectures Center at Syracuse University, the Cornell Theory Center, the Center for Advanced Computing Research at Caltech, and the South Australian Centre for Parallel Computing.

References

[1] S.A. Cuccaro, M. Mascagni and D.V. Pryor, Techniques for testing the quality of parallel pseudo-random number generators, in *Proc. of the 7th SIAM Conf. on Parallel Processing for Scientific Computing*, SIAM, Philadelphia, 1995, p. 279.

[2] A. De Matteis and S. Pagnutti, Controlling correlations in parallel Monte Carlo, *Parallel Computing* **21**, 73 (1995).

[3] Paul D. Coddington, Random Number Generators for Parallel Computers, *The NHSE Review*, `http://nhse.cs.rice.edu/NHSEreview/`, 1996 Volume, Second Issue.

[4] C. Koelbel *et al.*, *The High Performance Fortran Handbook*, MIT Press, Cambridge, Mass., 1993.

[5] S.L. Anderson, Random Number Generators on Vector Supercomputers and Other Advanced Architectures, *SIAM Rev.* **32**, 221 (1990).

[6] J. Apostolakis, P. Coddington and E. Marinari, New SIMD algorithms for cluster labeling on parallel computers, *Int. J. Mod. Phys. C* **4**, 749 (1993).

[7] S.-J. Bae, S.H. Ko and P.D. Coddington, Parallel Wolff Cluster Algorithms, *Int. J. Mod. Phys. C* **6**, 197 (1995).

[8] C.F. Baillie and P.D. Coddington, Cluster identification algorithms for spin models – sequential and parallel, *Concurrency: Practice and Experience* **3**, 129 (1991).

[9] C.F. Baillie and P.D. Coddington, Comparison of cluster algorithms for 2-D Potts models, *Phys. Rev.* **B 43**, 10617 (1991); P.D. Coddington and C.F. Baillie, Empirical relations between static and dynamic exponents for Ising model cluster algorithms, *Phys. Rev. Lett.* **68**, 962 (1992).

[10] K. Binder ed., *Monte Carlo Methods in Statistical Physics*, Springer-Verlag, Berlin, 1986; K. Binder and D.W. Heermann, *Monte Carlo Simulation in Statistical Physics*, Springer-Verlag, Berlin, 1988; H. Gould and J. Tobochnik, *An Introduction to Computer Simulation Methods, Vol. 2*, Addison-Wesley, Reading, Mass., 1988.

[11] R.P. Brent, Uniform random number generators for supercomputers, Proc. of the 5th Australian Supercomputer Conference, Melbourne, 1992, p. 95. Available at `ftp://nimbus.anu.edu.au/pub/Brent/rpb132.ps.gz`.

[12] T.-W. Chiu, Shift-register sequence random number generators on the hypercube concurrent computers, *Proc. of the 3rd Conference on Hypercube Concurrent Computers and Applications*, G. Fox ed., ACM Press, New York, 1988.

[13] P.D. Coddington, Analysis of Random Number Generators Using Monte Carlo Simulation, *Int. J. Mod. Phys. C* **5**, 547 (1994).

[14] P.D. Coddington and C.F. Baillie, Cluster algorithms for spin models on MIMD parallel computers, *Proc. of the 5th Annual Distributed Memory Computing Conference*, Charleston (April 1990), eds. D.W. Walker and Q.F. Stout, IEEE Computer Society Press, Los Alamitos, California, 1990.

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

[16] T. Filk, M. Marcu and K. Fredenhagen, Long range correlations in random number generators and their influence on Monte Carlo simulations, *Phys. Lett.* **B165**, 125 (1985).

[17] A.E. Ferdinand and M.E. Fisher, *Phys. Rev.* **185**, 832 (1969).

[18] G. Fox *et al.*, *Solving Problems on Concurrent Processors, Vol. 1*, Prentice-Hall, Englewood Cliffs, 1988.

[19] D.W. Heermann and A.N. Burkitt, *Parallel Algorithms in Computational Science*, Springer Verlag, Heidelberg, 1991.

[20] A. Hoogland, A. Compagner and H.W.J. Blöte, Smooth finite-size behavior of the three-dimensional Ising model, *Physica* **132A**, 593 (1985).

[21] F. James, A review of pseudorandom number generators, *Comp. Phys. Comm.* **60**, 329 (1990).

[22] C. Kalle and S. Wansleben, Problems with the random number generator RANF implemented on the CDC CYBER 205, *Comp. Phys. Comm.* **33**, 343 (1984).

[23] S. Kirkpatrick and E. Stoll, A very fast shift-register sequence random number generator, *J. Comput. Phys.* **40**, 517 (1981).

[24] D.E. Knuth, *The Art of Computer Programming Vol. 2: Seminumerical Methods* Addison-Wesley, Reading, Mass., 1981.

[25] P. L'Ecuyer, Random numbers for simulation, *Comm. ACM* **33:10**, 85 (1990).

[26] G.A. Marsaglia, A current view of random number generators, in *Computational Science and Statistics: The Interface*, ed. L. Balliard, Elsevier, Amsterdam, 1985.

[27] A. Milchev, K. Binder, D.W. Heermann, Fluctuations and lack of self-averaging in the kinetics of domain growth, *Z. Phys.* **B 63**, 521 (1986).

[28] J. Flower and A. Kolawa, Express is not just a message passing system: Current and future directions in Express, *Parallel Computing* **20**, 597 (1994).

[29] *Express Reference Manual*, *Express C User's Guide*, and *Express Fortran User's Guide*, Parasoft Corporation, Pasadena, CA, 1990.

[30] G. Parisi and F. Rapuano, Effects of the random number generator on computer simulations, *Phys. Lett.* **157B**, 301 (1985).

[31] S.K. Park and K.W. Miller, Random number generators: Good ones are hard to find, *Comm. ACM* **31:10**, 1192 (1988).

[32] O.E. Percus and J.K. Percus, Long range correlations in linear congruential generators, *J. Comput. Phys.* **77**, 267 (1988).

[33] R.H. Swendsen and J.-S. Wang, Nonuniversal critical dynamics in Monte Carlo simulations, *Phys. Rev. Lett.* **58**, 86 (1987).

[34] M. Flanigan and P. Tamayo, A parallel cluster labeling method for Monte Carlo dynamics, *Int. J. Mod. Phys. C* **3**, 1235 (1992).

[35] *CM Fortran User's Guide*, Thinking Machines Corporation, Reading, Mass., 1992.

[36] *MP Fortran User's Guide*, Maspar Corporation, 1992.

[37] *CM Scientific Software Library*, Thinking Machines Corporation, Reading, Mass., 1992.

[38] I. Vattulainen, T. Ala-Nissila and K. Kankaala, Physical models as tests of randomness, *Phys. Rev.* **E 52**, 3205 (1995).

[39] R.M. Ziff, Reduction of correlations in shift-register sequence random number generators using multiple feedback taps, unpublished.

[40] P. Grassberger, On correlations in "good" random number generators, *Phys. Lett. A* **181**, 43 (1993).

[41] U. Wolff, *Phys. Rev. Lett.* **62**, 361 (1989).

[42] S. Wolfram, Random Sequence Generation by Cellular Automata, *Adv. Appl. Math.* **7**, 123 (1986).

[43] CMMD Reference Manual, Thinking Machines Corporation, Cambridge, Mass., 1993.

[44] William Gropp, Ewing Lusk and Anthony Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, Mass., 1994.

[45] P.R. Bevington, *Data Reduction and Error Analysis for the Physical Sciences*, McGraw-Hill, New York, 1969.

[46] P.D. Coddington, S.-H. Ko, W.E. Mahoney and J.M. del Rosario, Monte Carlo Tests of Parallel Random Number Generators, in preparation.

[47] E.T. Dudewicz and T.G. Ralley, *The Handbook of Random Number Generation and Testing with TESTRAND Computer Code*, American Science Press, Columbus, Ohio, 1981.

[48] P.D. Coddington, Tests of random number generators using Ising model simulations, in *Proc. of the 1995 US-Japan Bilateral Seminar on New Trends in Computer Simulations of Spin Systems*, *Int. J. Mod. Phys.* **C 7**, 295 (1996).

[49] E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Rockville, Maryland, 1978; G. Brassard and P. Bratley, Algorithmics: Theory and Practice, Prentice Hall, Englewood Cliffs, N.J., 1988.