DHPC-???

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

9 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 models for which exact results are known. Preliminary results of these tests are presented for some commonly-used 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 [23, 20, 25, 32, 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 [21, 31, 16, 28, 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. However, to our knowledge, no adequate set of statistical tests has been proposed, let alone implemented, to test parallel random number 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 a few 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 as LFGs using XOR.

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

For more information, see one of the many books or review articles on random number generators [23, 20, 25, 32, 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 independent 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 these are not widely 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 widely used generators, 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]. Parallelization, particularly using leapfrog or sequence splitting, may amplify small correlations in the sequential generator. Great care needs to be taken in initializing the generators, so that the seeds are not correlated over different processors. Also, some generators have periods that may be adequate for current sequential workstations, but not for Teraflop supercomputers. 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 [23, 27, 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]. This is the usual approach in testing parallel generators. A parallel version of the spectral test for linear congruential generators has also been proposed [33]. However, very little work has been done on developing tests for parallel random number generators.

3.1 Physical Tests

In addition to standard statistical tests, it is useful to apply application-specific (or "physical") tests that are related to the various applications for which random numbers are 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 [22, 19, 15, 13, 38], percolation models [39], and random walks [40, 39, 38]. Generators that pass standard statistical tests have often been found to fail these application-specific tests. It is therefore important to use as wide a variety of these empirical tests as possible.

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. We have used some of their programs in testing parallel generators.

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 [10]. 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 method 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 algorithm [34] 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 10^5 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. The simulations are usually done at the critical point (or phase transition) of the model [10].

We have implemented the 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 [29, 30], CMMD [43], or MPI [44]. The data parallel programs were written in CM Fortran [36] or High Performance Fortran [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].

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, 35] 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.

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.

The simplest type of 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, and the error in the mean is easily computed using standard techniques like binning that take into account possible correlations in the measurements [10, 13].

For example, a physical test might consist of starting a random walk at the center of a square grid, counting what percentage of the time the random walk finishes on each of the four sides of the grid, and comparing the results with the expected exact value of 0.25. The result obtained for each side of the grid will be the mean value (average) over many independent random walks, and the statistical error is just the error in the mean, which is easily computed.

To check the quality of the random number generator, we simply compute the deviation $\Delta = (\mu - e)/\sigma$ between the computed mean value mu and the exact value e, as a multiple of the error in the mean (or standard deviation) 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. 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 test 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 several 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.

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. 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 a few different parallel random number generators using both of the approaches mentioned above – replicated sequential tests and fully parallel test programs. 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 a Thinking Machines CM-2 and CM-5, 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.

TBD - CHECK THIS AND INCLUDE NEW TEST, PARTICULARLY KO'S NEW STUFF FOR REPLICATED SEQUENTIAL TESTS.

The following parallel random number generators have been tested:

1. `CMF_RANDOM`, the parallel cellular automata generator used on the Connection Machine [36, 42].

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

3. `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.

4. `PRANDOM`, the parallel version of the standard Unix and C lagged Fibonacci generator `random`. We tested both the older (`PRANDOM #1`) and more recent (`PRANDOM #2`) versions implementated by Maspar. These differ in how the seed tables are initialized.

5. `PDRAND48`, the 48-bit C and Unix generator, parallelized using a leapfrog technique.

6. `PRANDOM`, the parallel version of the standard Unix and C Lagged Fibonacci generator `random`, used by Maspar.

7. `LECUYER`, L'Ecuyer's mixed LCG combining L(40014,0,2147483563) and L(40692,0,2147483399), using splitting [24, 26].

TBD - DESCRIBE REPLICATED AND PARALLEL TESTS.

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. Each simulation was 5×10^5 sweeps (Metropolis) or 2.5×10^5 sweeps (Swendsen-Wang and Wolff) of a 128×128 lattice at the critical point of the 2-d Ising model. The amount of random numbers generated for each of the 25 tests was of order 5×10^9, or of order 10^{11} in total. In some cases we have also tested the generators using alternate lattice sizes (64×64 and 256×256), which can probe possible correlations at different scales.

TBD - DISCUSSION OF RESULTS.

5 Conclusions

TBD - NEED TO ADD TO THIS BASED ON RESULTS

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.

Physical tests based on computational science applications have proven to be very useful in identifying problems in random number generators.

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 oudated 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.

Although a lot of research has been done on developing improved parallel random number generators, little is being done on developing rigorous empirical tests for such generators.

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.

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, 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.

[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), p. 1421.

[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.*, Chapter 12 of *Solving Problems on Concurrent Processors, Vol. 1*, (Prentice-Hall, Englewood Cliffs, 1988).

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

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

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

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

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

[24] P. L'Ecuyer, Efficient and portable combined random number generators, *Comm. ACM* **31:6** (1988) 742.

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

[26] P. L'Ecuyer and S. Côte; Implementing a random number package with splitting facilities, *ACM Trans. Math. Soft.* **17** (1991) 98.

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

[28] 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).

[29] J. Flower and A. Kolawa, Express is not just a message passing system: Current and future directions in Express, P597-614 *Parallel Computing* **20**, 597 (1994).

[30] *Express Reference Manual*, *Express C User's Guide*, and *Express Fortran User's Guide*, Parasoft Corporation, 2500 E. Foothill Blvd., Pasadena CA 91107 (1990).

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

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

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

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

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

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

Table 1: Results of Monte Carlo simulations of the 2-d Ising model using different 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. Numbers in bold type indicate results which would occur with a statistical probability of less than 0.001 for true random sequences.

		Energy			Specific Heat		
Lattice Size	Generator	SW	Wolff	Metrop	SW	Wolff	Metrop
64×64	CMF_RANDOM	-0.96	–	**-10.68**	0.60	–	**5.07**
		0.83	–	**8.75**	0.74	–	**2.72**
128×128	CMF_RANDOM	0.6?	–	-0.24	0.4?	–	-1.61
		0.7?	–	0.92	1.1?	–	1.49
256×256	CMF_RANDOM	–	–	**-7.13**	–	–	1.23
		–	–	**4.93**	–	–	**2.41**
128×128	PRANDOM #1	–	–	**53.04**	–	–	**-10.34**
		–	–	**201.83**	–	–	**17.24**
128×128	PRANDOM #2	–	–	1.90	–	–	**3.14**
		–	–	-2.25	–	–	**4.44**

Table 2: Results of Monte Carlo simulations of the 2-d Ising model using different random number generators on MIMD 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. Numbers in bold type indicate results which would occur with a statistical probability of less than 0.001 for true random sequences.

		Energy			Specific Heat		
Lattice Size	Generator	SW	Wolff	Metrop	SW	Wolff	Metrop
64×64	`CMSSL F(17,5,+)`	0.06	–	**-3.49**	0.79	–	1.62
		-0.30	–	1.04	1.19	–	1.22
128×128	`CMSSL F(17,5,+)`	–	-0.21	-1.97	–	0.99	-0.30
		–	0.17	1.00	–	1.23	0.52
128×128	`F(1279,1063,+)`	1.02	-0.35	0.96	1.05	0.60	0.63
		1.48	0.93	**2.35**	0.69	1.24	1.27
64×64	`PRAND`	–	–	**6.97**	–	–	**4.88**
		–	–	**-5.28**	–	–	**5.17**
128×128	`PRAND`	-1.88	–	**221.65**	-0.07	–	**-55.21**
		0.79	–	**10350**	0.85	–	**629.60**