1996

Tests of random number generators using Ising model simulations

Paul D. Coddington
Syracuse University, Northeast Parallel Architectures Center

SCCS-746

TESTS OF RANDOM NUMBER GENERATORS USING ISING MODEL SIMULATIONS

P. D. CODDINGTON

Northeast Parallel Architectures Center, Syracuse University
111 College Place, Syracuse, NY 13244, U.S.A.

ABSTRACT

Large-scale Monte Carlo simulations require high-quality random number generators to ensure correct results. The contrapositive of this statement is also true – the quality of random number generators can be tested by using them in large-scale Monte Carlo simulations. We have tested many commonly-used random number generators with high precision Monte Carlo simulations of the 2-d Ising model using the Metropolis, Swendsen-Wang, and Wolff algorithms. This work is being extended to the testing of random number generators for parallel computers. The results of these tests are presented, along with recommendations for random number generators for high-performance computers, particularly for lattice Monte Carlo simulations.

Keywords: Random Number Generators, Ising Model, Monte Carlo Simulation

1. Introduction

Random number generators are widely used for simulations in computational science and engineering, in particular for large-scale Monte Carlo simulations on high-performance computers, for which good randomness properties are essential. A poor random number generator can produce incorrect results from the simulation, which has been seen many times in the computational physics literature.[1,2,3,4,5,6,7] Monte Carlo simulations can therefore provide sensitive tests of the randomness properties of random number generators. In the past, a common test has been the Metropolis Monte Carlo simulation of the 2D Ising model,[8] for which exact results are known.[9] Recently it was found that the Swendsen-Wang and Wolff cluster algorithms can also provide very sensitive tests of random number generators.[5,7,10]

Large-scale Monte Carlo simulations are now commonly performed on high-performance parallel computers. Although many parallel algorithms for random number generation have been proposed, little testing has been done on these algorithms, which is a very dangerous situation. The many problems caused in the past by inadequate random number generators are likely to be repeated in a new generation of simulations using parallel computers, unless these generators are very carefully studied and tested.

In this paper, we summarize the results of Monte Carlo tests of random number generators on sequential computers,[7] and provide preliminary results on tests of random number generators for parallel machines.[11] We first present the different algorithms used for standard sequential random number generators, and the various methods used to parallelize these algorithms. We then briefly describe the Monte Carlo methods used to test the generators, and summarize the results of these tests. Finally, we discuss the various random number generators in the light of these results, and make recommendations on generators for use on high-performance computers, particularly for lattice Monte Carlo simulations.

2. Random Number Generators

2.1. *Linear Congruential Generators*

Probably the most commonly-used random number generators are linear congruential generators (LCGs).[12,13,14] The standard C and Unix generators `RAND` (32-bit precision) and `DRAND48` and `RANF` (48-bit precision) are of this type. Note that for 32-bit integers the period of these generators is at most 2^{32}, or of order 10^9. On modern workstations, capable of 10^7 – 10^8 floating point operations per second, this period can be exhausted in a matter of minutes, so generators with 48-bit or 64-bit precision must be used.

LCGs work very well for most applications but have two major defects. The first is that the least significant bits of the numbers produced are highly correlated, and a resultant "scatter-plot" of ordered pairs of random floating point numbers in the interval (0,1) shows regular lattice structure.[15,16,17,18] They are also known to have long-range correlations, in particular for intervals which are a power of 2.[1,3,19,20]

2.2. *Lagged Fibonnaci and Shift Register Generators*

Lagged Fibonacci generators (LFGs)[12,14,15] are becoming increasingly popular, since they offer a simple method of obtaining very large periods. The standard C and Unix generator `RANDOM` is of this type. Different versions of this generator are denoted by $\mathrm{F}(p, q, \odot)$, where p and q (the lags) refer to previous elements of the sequence that are combined using the binary arithmetic operation $\odot$, which can be $+, -, *$ or $\oplus$ (XOR). The period can be made arbitrarily large by increasing the lag, which also improves the randomness properties.[15,7]

Shift register (or Tausworthe) generators[12,15,21,22,23] can be considered as a special case of a lagged Fibonacci generator using XOR. Because XOR is such a simple operation, it gives the worst randomness properties of any operation for an LFG.[15,7] However this simplicity means that these generators are very fast and thus still popular in spite of their serious drawbacks.

The randomness properties of these generators can be greatly improved by using multiple "taps",[23,24] i.e. by combining four or more previous elements of the sequence, rather than two.

2.3. *Combined Generators*

Combining two different generators has been shown (both theoretically and empirically) to produce an improved quality generator in many circumstances.[15,17,25] L'Ecuyer[17] has shown how to additively combine two different 32-bit LCGs to produce a mixed generator which passes the scatter-plot test and has a long period of around 10^{18}, thus overcoming some of the drawbacks of standard LCGs. More recently L'Ecuyer *et al.*[26] have proposed 48-bit and 64-bit combined generators, with even larger periods and presumably better randomness properties.

Marsaglia has suggested combining a fast, simple Weyl (or arithmetic sequence) generator with a lagged Fibonacci generator, which is the basis for the popular `RANMAR` generator.[14,27]

3. Parallel Random Number Generators

3.1. *The Leapfrog Method*

Ideally we would like a parallel random number generator to produce the same sequence of random numbers for different numbers of processors, which makes the parallel code more portable and much easier to debug. A simple way of achieving this is for processor P of an N processor

machine to generate the sub-sequence $X_P, X_{P+N}, X_{P+2N}, \ldots$, so that the sequence is spread across processors in the same way as a deck of cards is dealt in turn to players in a card game. This is known as the leapfrog technique,[28,29] since each processor "leapfrogs" by N in the sequence. In order to use this method we need to be able to easily jump ahead by N in the sequence. This can be done for linear congruential generators,[28,29] combined LCGs[30,31] and shift-register generators.[32]

One would expect that a large period (at least 48-bit) parallel linear congruential generator using the leapfrog method should be adequate for most purposes. However there is a problem – linear congruential generators are known to have correlations between elements in the sequence that are a power of 2 apart. For many parallel or vector machines the number of processors or the vector length is a power of 2, and this is also often the case for the size of the arrays used in a simulation (e.g. grid size or lattice size). This means that the pseudo-random numbers generated on a given processor may be more strongly correlated than the sequence on a single processor. In fact this type of leapfrog linear congruential algorithm, when used on vector machines, has led to spurious results in some Monte Carlo calculations.[1,3]

3.2. *Sequence Splitting*

Another way of parallelizing linear congruential generators is to split the sequence into non-overlapping parts, each generated by a different processor.[30,31] This also requires the ability to jump ahead in the sequence by a given amount. For example, one could divide the period of the generator by the number of processors, and jump ahead in the sequence by this amount for each processor. A program for a generator of this type using sequence splitting of a combined LCG is given by L'Ecuyer and Côté.[31]

One disadvantage of this type of generator is that it does not produce the same sequence for different numbers of processors. However in a data parallel programming model (for example, as used by High Performance Fortran[33]) it is possible to split the sequence among "abstract processors", or distributed data elements, such that the sequences will be the same for any number of processors.[34]

3.3. *Independent Generators*

The simplest method for using lagged Fibonacci generators in parallel is to just run the same sequential generator on each processor, but with different initial lag tables (or *seed tables*).[35,36] In order for this method to work successfully, it is crucial that the seed tables on each processor are random and independent, since any initial correlations may be preserved and adversely affect the random numbers that are produced. It has also been observed that correlations between the seed tables on different processors can also be propagated throughout the sequences of generated numbers. This was a problem with the initial implementation of this type of generator in the standard software library for the Maspar (or DECmpp) parallel computer, which gave extremely poor quality pseudo-random numbers.

Unlike the sequence splitting generator, this method does not guarantee that there is no overlap between the sequences generated on different processors. However using large lags eliminates this problem to all practical purposes, since the probability of overlap will be completely negligible. This algorithm is used in the Connection Machine Scientific Software Library (CMSSL) routine `FAST_RNG`,[37] where the interface to the routine allows the user to specify the lags, so in principle the routine can be extremely good, although the CMSSL documentation suggests using lags which are much too small to give good randomness properties.

A deficiency of this method is that is does not produce the same sequence for different numbers of processors. However, this can be achieved by a simple variation of this method, which is also

provided in the CMSSL as the `VP_RNG` generator.[37] A different generator (i.e. a different lag table) is assigned to every abstract processor, rather than every processor. This makes the algorithm independent of the number of processors, however the memory requirement is likely to be exorbitant with this approach.

3.4. *Other Methods*

The cellular automata generator is a generalization of the shift register generator, based on cellular automata rules.[38] A parallel version is provided by Thinking Machines[39] and is one of the generators we have tested.

Anderson[40] has reviewed a number of different methods for generating random numbers on vector and parallel machines.

4. Monte Carlo Tests

Various random number generators were tested by using them for Monte Carlo simulation of the two dimensional Ising model.[8] This simple model has been solved exactly for a finite lattice,[9] so that values of the energy and the specific heat of the system calculated from the Monte Carlo simulation can be compared with the known exact values. Three different methods were used for the simulations: the Metropolis algorithm[8] which updates a single site of the lattice; the Swendsen-Wang algorithm[41] which forms clusters of sites to be updated collectively; and the Wolff algorithm[42] which updates a single cluster of sites. These methods use the random numbers in very different ways.

We have developed both sequential and parallel implementations of all three of these Monte Carlo methods, with both data parallel and message passing implementations of the parallel algorithms. The Metropolis algorithm is parallelized with standard domain decomposition methods, using a checkerboard or red/black updating scheme.[28] The message passing Swendsen-Wang program uses the local label propagation or self-labeling technique for parallel cluster labeling.[43,44,45] This program has been used for high-precision Monte Carlo studies of Ising and Potts spin models.[46] A different, data parallel version of the Swendsen-Wang algorithm was also used,[47] and both message passing and data parallel Wolff algorithms have been developed.[48]

Various parallel random number generators were tested using a Thinking Machines CM-2 and CM-5, Maspar MP-100, nCUBE/2, and Intel iPSC/860. The sequential generators were tested on a network of workstations. The message passing programs were run on 16 processors, except for the 32-processor CM-5. The results of the data parallel programs are dependent primarily 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. This work is still in progress and only preliminary results are given here. Other parallel random number generators will eventually be tested.[11]

For each random number generator, 25 independent simulation runs with different initial seeds were performed at the critical point of the 2-d Ising model.[8] A 16^2 lattice was used for the sequential algorithms, and a 128^2 lattice for the parallel algorithms. In some cases the generators were also tested using alternate lattice sizes, which can probe possible correlations at different scales. A number of the parallel generators passed tests for one lattice size and failed for another. The total amount of random numbers generated in the 25 tests was of order 10^{11} for both the sequential and parallel tests.

The sequential generators tested, and the statistical techniques used in testing the generators, are described in detail in a previous paper.[7] In addition, the following parallel random number generators were tested:

1. `CMF_RANDOM`, the parallel cellular automata generator used on the Connection Machine.[39,38]

2. `PRAND`, the standard 32-bit C and Unix generator `RAND`, parallelized using a leapfrog technique.[28]

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]

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. Results and Recommendations

The results of the tests are summarized in Table 1 for sequential generators and Table 2 for parallel generators. A check mark indicates the generator passed the particular tests, a cross means that it failed.

Generator	S-W	Wolff	Met	Generator	S-W	Wolff	Met
RCARRY	**X**	**X**	**X**	RANMAR	$\surd$	$\times$	$\surd$
SWC	$\surd$	**X**	$\surd$	F(4423,1393,+)	$\surd$	$\surd$	$\surd$
F(250,103,$\oplus$)	$\surd$	**X**	$\surd$	F(4423,1393,$\oplus$)	$\surd$	$\surd$	$\surd$
F(250,103,+)	$\surd$	**X**	$\surd$	F(218,95,39,11,$\oplus$)	$\surd$	$\surd$	$\surd$
RAND	$\surd$	**x**	**x**	F(55,24,16,8,+)	$\surd$	$\surd$	$\surd$
CONG	$\surd$	**x**	**x**	F(5,2,$*$)	$\surd$	$\surd$	$\surd$
SWCW	$\surd$	**x**	$\surd$	F(43,22,$*$)	$\surd$	$\surd$	$\surd$
F(1279,1063,$\oplus$)	$\surd$	**x**	**x**	RANECU	$\surd$	$\surd$	$\surd$
F(55,24,16,8,$\oplus$)	$\surd$	**x**	$\surd$	RAN2	$\surd$	$\surd$	$\surd$
F(1279,1063,+)	$\surd$	$\times$	$\surd$	DRAND48	$\surd$	$\surd$	$\surd$
F(2,1,$*$) + Weyl	$\surd$	$\times$	$\surd$	RANF	$\surd$	$\surd$	$\surd$

Table 1. Results of Monte Carlo simulations of the 2-d Ising model on a 16×16 lattice using different random number generators. **X** means the generator failed the test after 10^6 sweeps, **x** means it failed after 10^7 sweeps, $\times$ means it failed, and $\surd$ means it passed, after 5×10^7 sweeps (or 10^7 sweeps for the Swendsen-Wang algorithm).

The best sequential random number generators tested were the lagged Fibonacci generators using multiplication, linear congruential generators with at least 48-bit precision, and combined generators such as `RANECU`[17,14] (L'Ecuyer's combined LCG generator) and `RANMAR`[27,14] (Marsaglia's combined LFG and Weyl generator, although the lag used should be much greater than the value of 97 recommended by Marsaglia).

Many of the sequential generators, and most of the parallel generators, failed these Monte Carlo 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 for sequential computers,[12,13] and a similar problem is now occurring with generators for parallel and vector machines.

By the year 2000 supercomputers will have Teraflop (10^{12} floating point operations per second) performance, and a Teraflop-year of computation (3×10^{19} flops) will become realizable for such

Generator	S–W		Wolff		Metropolis		
	64^2	128^2	64^2	128^2	64^2	128^2	256^2
CMSSL FAST_RNG	$\surd$	X	$\surd$	$\surd$	X	$\surd$	–
F(1279,1063,+)	–	$\surd$	–	$\surd$	–	$\surd$	–
PRAND	–	$\surd$	–	–	X	X	–
CMF_RANDOM	$\surd$	$\surd$	–	–	X	$\surd$	X
PRANDOM #1	–	–	–	–	–	X	–
PRANDOM #2	–	–	–	–	–	X	–

Table 2. Results of Monte Carlo simulations of the 2-d Ising model using various lattice sizes for different parallel random number generators. X means the generator failed the test, and $\surd$ means it passed, after 5×10^5 sweeps for Metropolis or 2.5×10^5 sweeps for S-W and Wolff. A dash means the test has not yet been done.

problems as Monte Carlo simulation of lattice gauge theory and condensed matter physics.[49] Such large scale Monte Carlo simulations will exhaust the period (of roughly 10^{18}) of 64-bit LCGs or combined 32-bit LCGs. It will thus be necessary in the near future to move to very long period generators such as large-lag multiplicative LFGs or combined 64-bit LCGs.[26] These generators should have both the randomness properties and the extremely large period required for applications of the 21st century (the period of these generators is large enough to handle a Petaflop-age-of-the-universe computation).

Shift register generators and additive LFGs have been popular in the past because they were much faster than generators which use multiplication. However the difference in performance is greatly reduced on modern processors, and since generators using multiplication will generally have much better randomness properties, they should be used. It should also be noted that the speed of a random number generator is often irrelevant, since in most applications the amount of time spent generating the random numbers is insignificant compared to the rest of the calculation. The quality of the random numbers is usually far more important than the speed with which they are generated, so it's better to be slow than sorry.

The theoretical understanding of random number generators is rather limited, and no amount of statistical testing can ever determine the quality of a generator. It is therefore recommended that for any stochastic simulation, at least two very different random number generators are used (for example, a multiplicative LFG and a combined generator such as `RANECU`) and the results compared, in order to be confident that the random number generator is not introducing a bias.

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

Acknowledgements

The simulations were run on computers at the Northeast Parallel Architectures Center (NPAC) at Syracuse University and the Center for Advanced Computing Research at Caltech. Work supported in part by the Center for Research on Parallel Computation with NSF cooperative agreement No. CCR-9120008 and by the National Aeronautics and Space Administration under cooperative agreement No. NCCW-0027.

References

1. C. Kalle and S. Wansleben, *Comp. Phys. Comm.* **33**, 343 (1984).
2. G. Parisi and F. Rapuano, *Phys. Lett.* **157B**, 301 (1985).
3. T. Filk, M. Marcu and K. Fredenhagen, Phys. Lett. **B165**, 125 (1985).
4. A. Milchev, K. Binder, D.W. Heermann, *Z. Phys.* **B 63**, 521 (1986).
5. A.M. Ferrenberg, D.P. Landau and Y.J. Wong, *Phys. Rev. Lett.* **69**, 3382 (1992).
6. P. Grassberger, *Phys. Lett. A* **181**, 43 (1993).
7. P.D. Coddington, *Int. J. Mod. Phys.* **C 5**, 547 (1994).
8. 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).
9. A.E. Ferdinand and M.E. Fisher, *Phys. Rev.* **185**, 832 (1969).
10. I. Vattulainen, T. Ala-Nissila and K. Kankaala, *Phys. Rev. Lett.* **73**, 2513 (1994).
11. P.D. Coddington, J.M. del Rosario, S.-H. Ko and W.E. Mahoney, Monte Carlo Tests of Parallel Random Number Generators, in preparation.
12. D.E. Knuth, *The Art of Computer Programming Vol. 2: Seminumerical Methods* (Addison-Wesley, Reading, Mass., 1981).
13. S.K. Park and K.W. Miller, *Comm. ACM* **31:10**, 1192 (1988).
14. F. James, *Comp. Phys. Comm.* **60**, 329 (1990).
15. G.A. Marsaglia, in *Computational Science and Statistics: The Interface*, ed. L. Balliard (Elsevier, Amsterdam, 1985).
16. G.A. Marsaglia, *Proc. Nat. Acad. Sci.* **61**, 25 (1968).
17. P. L'Ecuyer, *Comm. ACM* **31:6**, 742 (1988).
18. H. Neiderreiter, *Bull. Amer. Math. Soc.* **84**, 957 (1978).
19. O.E. Percus and J.K. Percus, *J. Comput. Phys.* **77**, 267 (1988).
20. J. Eichenauer-Herrmann and H. Grothe, *Numer. Math.* **56**, 609 (1989).
21. R.C. Tausworthe, *Math. Comput.* **19**, 201 (1965).
22. S. Kirkpatrick and E. Stoll, *J. Comput. Phys.* **40**, 517 (1981).
23. S.W. Golomb, *Shift Register Sequences*, (Holden-Day, San Francisco, 1967).
24. R.M. Ziff, Reduction of correlations in shift-register sequence random number generators using multiple feedback taps, unpublished.
25. L.-H. Deng, E.O. George and Y.-C. Chiu, in *Proc. of the 1991 Winter Simulation Conference*, ed. B.L. Nelson *et al*, p. 1035.
26. P. L'Ecuyer, F. Blouin, and R. Couture, *ACM Trans. on Modeling and Computer Simulation* **3**, 87 (1993).
27. G.A. Marsaglia, *Stat. Prob. Lett.* **8**, 35 (1990).
28. G. Fox *et al.*, Chapter 12 of *Solving Problems on Concurrent Processors, Vol. 1*, (Prentice-Hall, Englewood Cliffs, 1988).
29. W. Evans and B. Sugla, in *Proc. of the 4th Conference on Hypercube Concurrent Computers and Applications*, ed. J. Gustafson, (Golden Gate Enterprises, Los Altos, CA, 1989), p. 415.
30. P. L'Ecuyer, *Comm. ACM* **33:10**, 85 (1990).
31. P. L'Ecuyer and S. Côté, *ACM Trans. Math. Soft.* **17**, 98 (1991).
32. S. Aluru, G.M. Prabhu and J. Gustafson, *Parallel Comput.* **18**, 839 (1992).
33. C. Koelbel *et al.*, *The High Performance Fortran Handbook*, (MIT Press, Cambridge, Mass., 1994).
34. P.D. Coddington, O. Odeyemi and C. Stoner, A Random Number Generator for High Performance Fortran, in preparation.
35. T.-W. Chiu, in *Proc. of the 3rd Conference on Hypercube Concurrent Computers and Applications*, ed. G. Fox, (ACM Press, New York, 1988), p. 1421.
36. W.P. Peterson, *J. Supercomput.* **1**, 327 (1988).
37. *CM Scientific Software Library*, (Thinking Machines Corporation, Reading, Mass., 1993).
38. S. Wolfram, *Adv. Appl. Math.* **7**, 123 (1986).
39. *CM Fortran User's Guide*, (Thinking Machines Corporation, Reading, Mass., 1994).
40. S.L. Anderson, *SIAM Rev.* **32**, 221 (1990).
41. R.H. Swendsen and J.-S. Wang, *Phys. Rev. Lett.* **58**, 86 (1987).

42. U. Wolff, *Phys. Rev. Lett.* **62**, 361 (1989).
43. P.D. Coddington and C.F. Baillie, in *Proc. of the 5th Annual Distributed Memory Computing Conference*, ed. D.W. Walker and Q.F. Stout, (IEEE Computer Society Press, Los Alamitos, California, 1990), p. 384.
44. C.F. Baillie and P.D. Coddington, *Concurrency: Practice and Experience* **3**, 129 (1991).
45. M. Flanigan and P. Tamayo, *Int. J. Mod. Phys.* **C 3**, 1235 (1992).
46. C.F. Baillie and P.D. Coddington, *Phys. Rev.* **B 43**, 10617 (1991); P.D. Coddington and C.F. Baillie, *Phys. Rev. Lett.* **68**, 962 (1992).
47. J. Apostolakis, P. Coddington and E. Marinari, *Int. J. Mod. Phys.* **C 4**, 749 (1993).
48. S.-J. Bae, S.H. Ko and P.D. Coddington, *Int. J. Mod. Phys.* **C 6**, 197 (1995).
49. P. Rodgers, *Physics World* (Feb. 1991) p. 13; S. Aoki *et al.*, *Int. J. Mod. Phys.* **C 2**, 829 (1991); K. Binder, *Int. J. Mod. Phys.* **C 3**, 565 (1992).