Evaluation of the IBM ESSL

Paul Coddington,
Northeast Parallel Architectures Center,
Syracuse University, Syracuse, NY 13244-4100
Email: paulc@npac.syr.edu

Chapter 16. Random Number Generation

Random number generators are widely used for simulation in science and engineering problems. Randomness can often be present in the formulation of the problem, especially for systems where the variation of parameters (perhaps in time or space) is not known exactly, but can be modeled probabilistically, for example quantum processes, random noise or perturbations, and random defects in solid state physics. Many problems also use solution techniques which are stochastic, for example Monte Carlo simulation, and stochastic optimization techniques such as simulated annealing or genetic algorithms. The number of applications which utilize random number generators is therefore quite large, especially in the computational sciences.

In some simulations, the quality of the random numbers used is not that important, as long as they are "fairly random". However in many of the problems for which random numbers are most heavily used, such as Monte Carlo simulation, the quality of the random number generator is crucial. A poor generator can produce incorrect results, and this has been seen many times in the scientific literature [14, 3, 21, 12, 8, 20, 7, 11, 5]. Also, many generators fail standard statistical tests [15, 13, 17, 22, 28].

None of the random number generators provided by the ESSL are good enough for large scale stochastic simulation. Some recommendations for improved generators are given below. A recent paper on research at NPAC on testing sequential random number generators is appended.

Vendors of parallel computers generally provide library routines for random number generators, and these range from reasonable (Thinking Machines) to very poor (Maspar). There has been quite a lot of research on developing algorithms for parallel random number generators, but very little has been done on developing and using methods for testing such generators. There are many statistical and Monte Carlo tests for checking sequential random number generators, some of which can be simply extended to apply to parallel generators (although very few have been). However new and better tests are required for parallel algorithms, for example to test for correlations between random numbers produced on different processors. To my knowledge no good set of tests has been proposed, let alone implemented, for parallel random number generators. This would be a very useful research project, since the lack of results means that it is difficult to compare the different generators and to recommend one as being any better than another. Some work in this area has been done at NPAC, but it is very preliminary and is not being actively continued due to lack of funding and manpower.

1) The SURAND linear congruential generator

SURAND is a standard 32-bit linear congruential generator. This generator has good randomness properties, but its period is too small. The period of SURAND is $2^{31}-1$, or around 2×10^9, and can be exhausted on an RS/6000 in a few minutes. This generator is therefore completely inadequate for most scientific work. Although in the "Use Considerations" section of the documentation the short period is alluded to, the actual value of the period is not stated. I recommend that this generator be kept purely for historical purposes, and users should be strongly warned against using it.

A longer period linear congruential generator should be provided. This could be one of the 48-bit generators recommended in Refs. [5, 9] (for example the standard C and Unix generator DRAND48), or preferably a generator with even larger modulus, for example the generator with multiplier 13^{13} and modulus 2^{59} used in the Numerical Algorithms Group (NAG) library. Another generator which could be provided is that of L'Ecuyer, which combines two 32-bit linear congruential generators into a generator which has a period almost as long as a 64-bit generator, and probably better randomness properties [16].

On a machine with 64-bit integer arithmetic, these algorithms will be just as fast as the 32-bit generators, but the period and randomness properties are much better. On machines with 32-bit integer arithmetic, multiple-precision arithmetic will be required so the generator will be slower than a 32-bit generator. However for most simulations the proportion of the overall run-time spent calling the random number generator is very small, and good randomness properties are far more important than speed. If users are really worried about speed, then they will presumably be producing enough random numbers to exhaust the period of the 32-bit generator anyway.

The syntax for the call to the linear congruential generator is standard and adequate. However for a generator using more than 32-bit integers, the initial seed would have to be specified by an array rather than a single number, as in DRAND48.

2) The normally distributed random number generator SNRAND

For the reasons given above, this generator should not call SURAND, but rather a longer period linear congruential generator. The method used to create normally distributed random numbers, and the syntax for the call to the routine, are standard and adequate.

3) The very long period shift register generator

I deduce from the quoted period that the major lag of this generator is 1279, however the lags (p, q) should be stated explicitly in the documentation. It is pleasing to see such a large lag used, however even so, this generator shows demonstrably non-random behavior in some Monte Carlo applications [5, 7, 11]. Also, shift register generators of this kind are known to be inferior to lagged Fibonacci generators with the same lag (and thus the same memory requirement) which use addition, subtraction, or multiplication instead of exclusive OR (XOR) [5, 19]. I therefore see no reason to use them, except perhaps for their simpler application to vector and parallel processing (see the following section for more details).

I recommend that the XOR operation be replaced by either subtraction modulo 1 on floating point numbers in [0,1), or even better, multiplication on odd integers (modulo 2^{32}, or some large integer) [5, 19]. This will dramatically improve the quality of the generator for little or no cost in speed. If multiplication is used, a lag of 1279 is adequate. If subtraction is used, a larger lag is necessary to ensure adequate randomness (I would suggest $p = 9689, q = 4187$).

The calling syntax seems OK, although I am not sure why the "work area" *aux* has to be an "array of (at least) length 10000". The only values which need to be stored are those in the lag table (the array of previous values which are needed to compute the next value), which has length p.

Parallel algorithms

Quite a lot of work has been done on vectorizing random number generators, and a number of parallel random number generators have also been proposed. However there does not appear to be any consensus on which is the best method, mainly because very little work has been done on testing and comparing parallel random number generators.

Implementing a good parallel random number generator is quite difficult. Clearly it needs to be based on a good sequential generator, however there is the additional problem of possible correlations between random number sequences on different processors. Another difficulty is that it may be beneficial (for example, for debugging purposes) to produce the same sequence of random numbers for different numbers of processors, and in particular, the same sequence as on a sequential machine. This can be done, although generally the types of algorithms which allow this functionality are not the best possible methods.

Linear congruential and shift register generators can be parallelized or vectorized in this way, because it is known in these cases how to jump ahead in the sequence. For a sequence X_i of random numbers, processor P of an N processor machine can therefore generate the sub-sequence $X_P, X_{P+N}, X_{P+2N}, \ldots$ This is known as the "leapfrog" technique (see Refs. [6, 10] for the linear congruential case, Ref. [1] for the shift register case).

One would expect that a large period (i.e. at least 48-bit) parallel linear congruential generator using the leapfrog method should be fine for most purposes. However there is a problem – linear congruential generators are known to have correlations between elements in the sequence which are a power of 2 apart [8, 23]. Since for most parallel machines the number of processors is a power of 2, this means that the 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 a vector machine, has given spurious results in some Monte Carlo calculations [14, 8]. It is possible that this problem could be avoided by leapfrogging by $N+1$ rather than N, thus avoiding power of 2 correlations, but this would mean that the parallel algorithm no longer produces the same sequence of numbers as the sequential algorithm.

The leapfrog shift register method could work well, but only if the lag is extremely large ($p = 9689, q = 4187$ is probably adequate). Shift register algorithms probably also suffer from long range correlations, however this is not known theoretically, and good statistical tests are lacking for this generator.

Another way of parallelizing linear congruential generators is to split the sequence into non-overlapping parts, each of which is generated by a different processor. This also utilizes the fact that this type of generator allows you to easily calculate an arbitrary element of the sequence. One can therefore simply divide the period of the generator by the number of processors, and jump ahead in the sequence by this amount for each processor [6, 18]. This requires that the period of the generator be very large. A 32-bit generator is completely inadequate, and even a 48-bit generator may not be good enough on a fast massively parallel machine. A good 64-bit generator should work well, but perhaps better would be to use the generator of L'Ecuyer [16]. A program for a parallel generator of this type using sequence splitting is given in Ref. [18]. The only disadvantages of this type of generator are that it does not produce the same sequence on different numbers of processors, and it is relatively slow on a processor which does not have 64-bit integer arithmetic.

The simplest method of 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") [4, 24]. The lag tables on each processor should be initialized using a *different* parallel generator, for example a leapfrog or sequence splitting linear congruential generator. This should guarantee (especially for a generator with a long lag) statistical independence between processors. This is probably the best parallel random number generator known. Its only known deficiency is that is does not produce the same sequence for different numbers of processors. It is used in the Connection Machine Scientific Software Library (CMSSL) [27], where the interface to the routine allows the user to specify the lags, so in principle the routine can be extremely good, however in the documentation they suggest using lags which are much too small.

There are two methods of parallelizing lagged Fibonacci generators which will produce the same sequence for different numbers of processors, which is useful for debugging. However one uses too much memory, and the other has too much communication.

The first method, which is also provided in the CMSSL, is to assign a different lag table to every *virtual processor* (i.e. distributed array element), rather than every processor [27]. It is thus independent of the number of processors, however for a large lag table the memory requirement may be exorbitant. If multiplication is used in the lagged Fibonacci generator, a small lag will still give good results. This generator could therefore be useful for debugging small scale problems.

The second method is straightforward vectorization of the basic lagged Fibonacci algorithm. As long as the vector length is smaller than both p and q, a vector of new results can be obtained independently [2, 25]. This parallelism can also be utilized if the elements of the vector are distributed over processors. The problem with using this method on a distributed memory machine rather than a vector or shared memory machine, is that the results are then not distributed properly to be used for the next update, and substantial communication is required. This should be contrasted with the other algorithms described above, for which *no* communication is required.

Other parallel random number generators have been proposed and implemented, for example the cellular automata generator provided by Thinking Machines [29, 26], however they do not have a sound enough theoretical background, or enough testing done on them, to be trusted.

In summary, I would recommend providing three parallel generators:

1. A linear congruential generator or combined (L'Ecuyer) generator, preferably one which is used in the sequential library, using splitting to break up the sequence of random numbers among processors.

2. A parallel lagged Fibonacci generator, the same as provided in the sequential library, but with a different initial lag table on each processor, initialized using the parallel linear congruential generator.

3. A parallel generator which produces the same sequence of results for different numbers of processors (and on a sequential machine), for debugging purposes. A leapfrog algorithm (either linear congruential or shift register) is the simplest to implement, and should work fine for debugging purposes, however it should be clearly stated that this algorithm should only be used for debugging, and not for data collection, since their randomness properties are suspect.

References

[1] S. Aluru, G.M. Prabhu and J. Gustafson, A random number generator for parallel computers, *Parallel Comput.* **18** (1992) 839.

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

[3] M.N. Barber *et al.*, *Phys. Rev.* **B 32** (1985) 1720;

[4] 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.

[5] P.D. Coddington, Analysis of Random Number Generators Using Monte Carlo Simulation, *Int. J. Mod. Phys. C*, to appear. NPAC technical report SCCS-526.

[6] W. Evans and B. Sugla, Parallel random number generation, *Proc. of the 4th Conference on Hypercube Concurrent Computers and Applications*, (Golden Gate Enterprises, Los Altos, CA, 1989), p. 415.

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

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

[9] G.S. Fishman, Multiplicative congruential random number generators with modulus 2^β: an exhaustive analysis for $\beta = 32$ and a partial analysis for $\beta = 48$, *Math. Comp.* **54** (1990) 331.

[10] G. Fox *et al.*, Chapter 12 of *Solving Problems on Concurrent Processors, Vol. 1*, (Prentice-Hall, Englewood Cliffs, 1988).

[11] P. Grassberger, On correlations in “good” random number generators, Wuppertal University preprint WUB 93-03.

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

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

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

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

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

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

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

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

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

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

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

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

[24] W.P. Peterson, Some vectorized random number generators for uniform, normal, and Poisson distributions for the CRAY X-MP, *J. Supercomput.* **1** (1988) 327.

[25] W.P. Peterson, Lagged Fibonacci series random number generators for the NEC SX-3, *Int. J. High Speed Comput.*, to appear.

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

[27] *CM Scientific Software Library*, (Thinking Machines Corporation, Reading, Mass., 1992).

[28] I. Vattulainen *et al.*, A comparative study of some pseudorandom number generators, University of Helsinki preprint HU-TFT-93-22, hep-lat 9304008.

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