1997

Random Number Generators for Parallel Computers

Paul D. Coddington
Syracuse University, Northeast Parallel Architectures Center

Random Number Generators for Parallel Computers

Paul D. Coddington

Northeast Parallel Architectures Center, 111 College Place,
Syracuse University, Syracuse, NY 13244-4100, U.S.A.

paulc@npac.syr.edu

Version 1.1
28 April 1997

Abstract

Random number generators are used in many applications, from slot machines to simulations of nuclear reactors. For many computational science applications, such as Monte Carlo simulation, it is crucial that the generators have good randomness properties. This is particularly true for large-scale simulations done on high-performance parallel computers. Good random number generators are hard to find, and many widely-used techniques have been shown to be inadequate. Finding high-quality, efficient algorithms for random number generation on parallel computers is even more difficult. Here we present a review of the most commonly-used random number generators for parallel computers, and evaluate each generator based on theoretical knowledge and empirical tests. In conclusion, we provide recommendations for using random number generators on parallel computers.

Outline

This review is organized as follows:

A brief summary of the findings of this review is first presented, giving an overview of the use of parallel random number generators and a list of recommended algorithms.

Section 1 is an introduction to random number generators and their use in computer simulations on parallel computers.

Section 2 is a summary of the methods used to test and evaluate random number generators, on both sequential and parallel computers.

Section 3 gives an overview of the main algorithms used to implement random number generators on sequential computers, provides examples of software implementations of the algorithms, and states any known problems with the algorithms or implementations.

Section 4 gives a description of the most common methods used to parallelize the sequential algorithms, provides examples of software implementing these algorithms, and states any known problems with the algorithms or implementations on current high-performance computers.

Section 5 provides an overview our findings and gives recommendations for the use of random number generators on parallel computers.

Section 6 is a glossary of terms related to random number generators.

Finally, a list of the references cited in this article is provided.

Summary

Random number generators use iterative deterministic algorithms for producing a sequence of pseudo-random numbers that approximate a truly random sequence. Ideally the sequence should be uniformly distributed, uncorrelated, reproduceable, portable, easily changed by adjusting an initial seed value, easily split into many independent subsequences, have a large period of repetition, pass all empirical tests for randomness, and be generated rapidly using limited computer memory.

Parallel random number generators should in addition have no correlations between the sequences on different processors, produce the same sequence for different numbers of processors, and not require any data communication between processors.

Developing random number generators that satisfy all of these requirements is a very difficult problem, particularly for parallel computers.

The main algorithms used for sequential random number generators are the following:

- **Linear congruential generators** – these work well if the parameters are properly chosen, the modulus is a prime number, and the state used is at least 48 bits, and preferably 64 bits. Do not use 32-bit versions.

- **Lagged Fibonacci generators** – implementations using multiplication are the best, addition or subtraction can be used if speed is a major concern, XOR should not be used. The lags used must satisfy certain requirements, and should be as large as possible, with the largest lag being at least 127 for multiplication and 1279 for addition, and preferably much larger. Care must be taken in initializing the seed tables, to ensure the entries are random and uncorrelated.

- **Shift register generators** – these are not recommended since they have comparatively poor randomness properties.

- **Combined generators** – combinations of two linear congruential generators, or a lagged Fibonacci generator and a linear congruential or other generator, work well in practice if two good generators are used.

The main techniques used for parallelizing random number generators involve distributing the sequences of random numbers produced by a sequential generator among the processors in the following different ways:

- **Leapfrog** – The sequence is partitioned among the 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** – For some generators, the initial seeds can be chosen in such a way as to produce long period independent subsequences on each processor.

Random number generators, particularly for parallel computers, should not be trusted. It is **strongly** recommended that all simulations be done with two or more different generators, and the results compared to check whether the random number generator is introducing a bias.

On a sequential computer, good generators to use are:

- a multiplicative lagged Fibonacci generator with a lag of at least 127, and preferably 1279 or more;
- a 48-bit or preferably 64-bit linear congruential generator, that performs well in the Spectral Test and has a prime modulus;
- a 32-bit (or more) combined linear congruential generator, with well-chosen parameters, such as those recommended by L'Ecuyer;
- if speed is really crucial, an additive lagged Fibonacci generator with a lag of at least 1279 and preferably much greater, and possibly combined with another generator, as in `RANMAR`, or using 3 or more lags rather than 2.

All of the parallel random number generators covered in this review have some limitations or possible problems. Recommended generators to use on a parallel computer are:

- Combined linear congruential generators using sequence splitting;
- Lagged Fibonacci generators using independent sequences, with careful initialization to ensure the seed tables on each processor are random and uncorrelated.
  If you do not require the same results for different numbers of processors, a multiplicative or additive generator with a large lag can be used, with different lag tables on each physical processor.
  Otherwise a different lag table is used on each abstract processor, requiring a small lag due to memory constraints, in which case we recommend using multiplication rather than addition.

Some software implementing these recommended generators is available from the National HPCC Software Exchange (NHSE).

More work needs to be done on developing better random number generators for parallel computers, and subjecting these generators to more thorough empirical testing.

1 Introduction

Random number generators are widely used for simulations in computational science and engineering. Randomness is often present in the formulation of the problem, for example random noise or perturbations, and quantum processes. In addition, many algorithms are probabilistic, for example Monte Carlo simulation and stochastic optimization techniques such as simulated annealing or genetic algorithms. Random number generators are also used in many other applications, from slot machines to cryptography.

Since "random" numbers are in practice computed using deterministic algorithms, these are more accurately called pseudo-random number generators. In some simulations, the quality of the pseudo-random numbers (how closely they resemble truly random sequences) is not that important. 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. This is especially true in large-scale simulations on parallel supercomputers, which consume huge quantities of random numbers, and require parallel algorithms for random number generation.

As noted in a number of previous review articles [1, 2, 3, 4, 5, 6, 7], random number generators provided by computer vendors or recommended in some papers and computer science texts have often been of poor quality. Even generators that perform well in standard statistical tests for randomness have sometimes proven to be unreliable for certain applications, particularly in Monte Carlo simulations [8, 9, 10, 11, 12, 13, 14, 15, 16, 17]. The many problems caused in the past by inadequate random number generators on sequential and vector computers are likely to be repeated in a new generation of simulations using parallel computers, unless parallel random number generators are very carefully studied and tested, and the best algorithms made readily available to users.

The aim of this review is to provide up-to-date information and recommendations concerning the use of random number generators on parallel computers. A summary of the current state-of-the-art in the development of random number generators for parallel computers is presented, along with the results of theoretical evaluations and empirical tests of these generators. For each of the common algorithms used in parallel random number generators, we give examples of software implementations of the algorithm, and state any known problems with the algorithm or the implementation of the generator on current high-performance computers.

Since this review is aimed primarily at users who may not be interested in the theoretical details of the various algorithms for generating random numbers, many of the technical details have been omitted in order to keep the presentation as simple as possible. More detailed information can be found in a number of other reviews [1, 2, 3, 4, 5, 6, 7].

2 Testing and Evaluation of Random Number Generators

2.1 Requirements for Sequential Random Number Generators

Ideally a pseudo-random number generator would produce a stream of numbers that

1. are uniformly distributed,
2. are uncorrelated,
3. never repeats itself,
4. satisfy any statistical test for randomness,
5. are reproduceable (for debugging purposes),
6. are portable (the same on any computer),
7. can be changed by adjusting an initial "seed" value,
8. can easily be split into many independent subsequences,
9. can be generated rapidly using limited computer memory.

In practice it is impossible to satisfy all these requirements exactly. Since a computer uses finite precision arithmetic to store the state of the generator, after a certain period the state must match that of a previous iteration, after which the generator will repeat itself. Also, since the numbers must be reproduceable, they are not truly random, but generated by a deterministic iterative process, and therefore cannot be completely uncorrelated.

For practical purposes, we require that the period of repetition of the sequence be much larger than the number of pseudo-random numbers that might be used in any application, and that the correlations be small enough that they do not noticeably affect the outcome of a computation. The first requirement can be determined fairly easily, knowing the power of the computer to be used. The second is extremely difficult to ascertain, and will generally be application-dependent.

It has often been the case that the speed of random number generators has been overly emphasized, to the detriment of the quality of the generators. A fast generator requires a minimal number of very simple operations, and it is this simplicity that often leads to problems with the quality of such generators. For most applications the speed of a random number generator is not even a concern, since the amount of time spent generating the random numbers is insignificant compared to the rest of the calculation. Users who yearn for super-fast random number generators usually have applications that spend most of their time generating random numbers, and require a huge number of them. These types of applications are often the ones that are most sensitive to the quality of the generator, in which case it would seem prudent to sacrifice a little speed for much better randomness properties. In using a random number generator, it's usually better to be slow than sorry.

The speed of a random number generator can usually be increased by allowing the routine to return an array of values, rather than a single value. This is clearly advantageous for a vector or parallel implementation, however it is also usually true for a sequential implementation, since it amortizes the cost of the function call to the random number generator.

2.2 Tests for Sequential 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 tests of this kind are available [1, 2, 18].

In addition to standard statistical tests, it is useful to apply application-specific tests that are more relevant to some of the various applications for which random numbers are used. As with the statistical tests, these tests generally compare the results obtained using a pseudo-random number generator with known exact results that would occur if the numbers were truly random. Tests of this kind include Monte Carlo simulation [19, 20] of exactly solvable systems such as the two dimensional Ising model [10, 21, 13, 15, 16, 17], simulations of percolation models [22], and random walks [14, 22, 17]. Generators that pass standard statistical tests have sometimes been found to fail these application-specific tests. It is therefore important to use as wide a variety of empirical tests as possible. Any application can in principle be used to test random number generators, by comparing results obtained with two different generators [8, 9, 10].

2.3 Requirements for Parallel Random Number Generators

In addition to the requirements for an ideal sequential random number generator given in Section 2.1, a random number generator for a parallel computer should ideally have the following additional properties:

1. The generator should work for any number of processors.
2. The sequences of random numbers generated on each processor should all satisfy the requirements of a good sequential generator, e.g. they should be uniformly distributed, uncorrelated, and have a large enough period.
3. There should be no correlations between the sequences on different processors.
4. The same sequence of random numbers should be produced for different numbers of processors, and for the special case of a single processor. This is beneficial for debugging purposes, and is a requirement for some parallel languages such as High Performance Fortran (HPF) [53] (which provides a random number generator as an intrinsic function, as does Fortran 90).
5. The algorithm should be efficient, which in practice means there should be no data movement between processors. Thus, after the generator is initialized, each processor should generate its sequence independently of the other processors.

As with the ideal sequential generator, in practice it is not feasible to meet all these requirements. Our goal is to find a parallel random number generator that measures up to these ideals as well as possible. For example, we may not be able to say for sure that there are no correlations between sequences on different processors, but we can at least ensure that there is no overlap between the sequences, and if possible try to minimize any correlations.

Note that we will use the word “processor” to refer to a single abstract processor, which may correspond to a physical processor, a process, a thread of control within a process, or an array element within a data parallel language such as HPF.

2.4 Tests for Parallel Random Number Generators

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.

There has been quite a lot of research on developing algorithms for parallel random number generators, but very little work has been done on developing and applying methods for testing such generators. Not many rigorous mathematical results are known about the properties of parallel random number generators, so stringent and varied empirical tests are vital.

The many standard statistical tests for checking the randomness properties of sequential generators can be applied to parallel generators, by testing the random number streams on each processor, and from all processors combined. This is the usual approach in testing parallel generators. However, new techniques 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 [25, 26]. Thus far, very little work has been done in this area.

A good empirical test of parallel random number generators is to use them with parallel implementations of the Monte Carlo algorithms used for simulating the two dimensional Ising model, which have proven to be very effective at testing sequential generators. Such tests have been done for a number of parallel generators [23, 24].

3 Random Number Generators

In this section we introduce the most common algorithms used in random number generators – the linear congruential, lagged Fibonacci, and shift register generators, and techniques which combine two or more of these generators. More detailed information can be found in a number of other review articles [1, 2, 3, 4, 5, 6, 7].

3.1 Linear Congruential Generators

Probably the most commonly-used random number generators are linear congruential generators (LCGs) [37, 1, 3, 5]. The standard C and Unix generators `RAND` (32-bit precision), `DRAND48` and `RANF` (48-bit precision) are of this type. LCGs produce a sequence X_i of random integers using the relation

$$X_i = (a * X_{i-1} + c) \mod M, \tag{1}$$

where a is known as the multiplier, M is the modulus, and c is an additive constant that may be set to zero. The parameters (a, c, M) must be chosen carefully to ensure a large period and good uniformity and randomness properties. The maximum period is M for suitably chosen parameters ($M - 1$ if $c = 0$) [1]. Standard random number generators return a floating point number x_i in the interval [0,1), which can be obtained by dividing X_i by M.

LCGs work very well for most applications but are known to have some major defects. The main problem is that the least significant bits of the numbers produced are correlated, and a "scatter-plot" of ordered tuples $(x_i, x_{i+1}, \ldots)$ of random floating point numbers plotted in the unit hypercube shows regular lattice structure [2, 27, 32, 33, 7]. This problem becomes worse in higher dimensions, which may affect some high-dimensional simulations. The problem of lattice structure can be quantified using the Spectral Test [1]. Examples of LCGs with good parameters that perform well in the Spectral Test are given in Refs. [1, 40, 15, 41].

Another problem is that many commonly-used LCGs (including `DRAND48` and `RANF`) use a modulus M that is a power of 2, since it is fast and convenient to implement this on a computer. However this approach produces highly correlated low-order bits [1, 4, 7], as well as long-range correlations for intervals that are a power of 2 [8, 11, 34, 35, 36]. This can cause problems for certain types of simulations, for example when using a hypercubic grid with a size that is a power of 2, or for applications that expect the lower-order bits to be random. To avoid these problems, it is best to use a modulus that is prime rather than a power of 2, however for many applications requiring single-precision floating point pseudo-random numbers (for which the low-order bits are irrelevant), `DRAND48` and `RANF` are quite adequate, and are often useful for checking the results obtained using other generators.

The effects of these regularities can be decreased by increasing the precision of the generators [2], for example by using 64-bit rather than 32-bit numbers. However there are practical limits to this approach – if M is greater than machine precision, then much slower multi-precision arithmetic must be used, so in practice the precision cannot be made arbitrarily large. This means that 48-bit and 64-bit LCGs can be quite slow on 32-bit computers, however many high-performance computers now use 64-bit processors, which should become standard in the near future. In that case, the speed of these generators is not a problem for most applications.

Note that for 32-bit integers the period of these generators is at most 2^{32}, or of order 10^9. On current processors capable of 10^8 floating point operations per second, this period can be exhausted in seconds, so higher precision (48-bit or more) generators should be used. For long simulations on high-performance computers, even 48-bit generators may have too small a period.

Despite their known problems, large precision LCGs with well-chosen parameters appear to work very well for all known applications, at least on sequential computers. However we will see later that there are problems with implementations of LCGs on parallel computers if the modulus is a power of 2.

Multiple recursive generators (MRG) [1, 5, 6, 41], generalize LCGs by using a recurrence of the form

$$X_i = (a_1 X_{i-1} + a_2 X_{i-2} + \ldots + a_k X_{i-k} + b) \mod M \tag{2}$$

for a given value of k. Choosing $k > 1$ will increase the time taken to generate each number, but will greatly improve the period and randomness properties of the generator [6, 41]. Some practical implementations have been provided for $k = 2$ [41].

3.2 Lagged Fibonnaci Generators

Lagged Fibonacci generators (LFGs) [1, 5, 2] are becoming increasingly popular, since they offer a simple method for obtaining very long periods, and they can be very fast. The standard C and Unix generator `RANDOM` is of this type. The sequence is defined by

$$X_i = X_{i-p} \odot X_{i-q} \tag{3}$$

which we will denote by $\mathrm{F}(p, q, \odot)$, where p and q are the lags, $p > q$, and $\odot$ is any binary arithmetic operation, such as addition or subtraction modulo M, multiplication modulo M, or the bitwise exclusive OR function (XOR). The arithmetic operations are done modulo any large integer value M, or modulo 1 if the X's are represented as floating point numbers in the interval [0,1), as can be done if the operation is addition or subtraction. Multiplication must be done on the set of odd integers. The modulus function is not required when using XOR. This method requires storing the p previous values in the sequence in an array called a *lag table*.

As with the LCGs, it is important that the parameters (p, q, M) be carefully chosen in order to provide good randomness properties and the largest possible period. If M is taken to be 2^b (i.e. the X's have b-bit precision), the maximal period is obtained if the lags are the exponents of a primitive polynomial [1, 28]. In that case the period is $2^p - 1$ for XOR, $(2^p - 1)2^{b-1}$ for addition and subtraction, and $(2^p - 1)2^{b-3}$ for multiplication [28, 7, 1, 2, 31]. Tables of suitable lags are available in the literature [1, 28, 30, 29, 31]. An advantage of this generator is that the period can be made arbitrarily large by just increasing the lag p. This also improves the randomness properties [2, 15], since smaller lags mean higher correlations between the numbers in the sequence. The low-order bits can have particularly poor randomness properties if small lags are used.

Empirical tests have shown that the randomness properties of LFGs are best when multiplication is used, with addition (or subtraction) being next best, and XOR being by far the worst [2, 15, 16, 17]. This is intuitively reasonable in that multiplication mixes the bits in two numbers much more than addition, which is in turn much better than XOR.

LFGs using addition (or subtraction) are the most popular because they are very simple and very fast. All the computation can be done in floating point, which avoids a costly integer to floating point conversion, and large periods can be obtained on 32-bit processors without having to use slow multi-precision arithmetic. Each pseudo-random number can be generated with just a single floating point addition and a modulus operation.

Great care needs be taken in choosing the lags for this type of generator. Many implementations use (or recommend in the documentation) lags that are much too small to give adequate randomness properties, even though it has been known for many years that additive LFGs fail some standard statistical tests for very small lags (such as $p = 17$) [2], and that increasing the lag improves the randomness properties of the generator. More recently, it was shown that even lags on the order of hundreds can give incorrect results in a number of tests based on common applications such as Monte Carlo simulation, percolation and random walks [14, 22, 15, 16, 17]. Unfortunately this information seems not to have percolated through the mathematical and computational science community, and it is still extremely rare to see a lag of greater than 1000 recommended for an additive LFG. We recommend using (p, q) of at least (1279,1063), and preferably much larger values. The standard Unix generator `RANDOM` is an additive LFG with a default lag of 31, which is much too small, however it is possible to initialize it with a larger lag.

Setting a minimum acceptable lag is obviously a moving target, since computers are continually becoming faster, allowing for more stringent randomness tests that use more random numbers. The recommendations given here are based on the results of the current set of application-specific tests mentioned above, using currently available computers. These values may not be adequate for future applications that use Teraflop computers. In any simulation, the largest feasible lag should be used, and the results should always be checked using a different generator.

Small lags were necessary in the past because of limited computer memory, however on current high-performance computers (and even personal computers), the additional memory requirement of a lag table with a few thousand entries can easily be handled. However we will see later that memory constraints may be a problem in implementing this algorithm on parallel computers for some applications. The choice of the lag may affect the speed of the generator, depending on the type of computer used. For example, if vector processors are used, a larger lag may improve performance, since the vector lengths are longer. If a scalar processor with limited cache memory is used, having a very large lag may cause cache misses and reduce the performance.

Multiplicative LFGs have seen little use, which is somewhat surprising considering their excellent randomness properties and extremely long period. Although slower than additive LFGs, they are just as fast as 32-bit LCGs, and much faster than LCGs that require multiple-precision arithmetic. Multiplicative LFGs can also be used with much smaller lags than for additive LFGs. Many different tests are failed by additive LCGs with small lags (less than 100) [2, 13, 14, 15, 16, 17], however no currently published results in any test show failure of a multiplicative LFG for a lag as small as 17. However we recommend using a lag of at least 100, to ensure a large period and better randomness properties.

One of the obstacles in implementing multiplicative LFGs is handling the possible overflow of the multiplication. In most languages a portable implementation would require multiple precision arithmetic, which would be quite slow. However if the C programming language is used, and the X's are specified to be of type `unsigned int`, then the language specification for the multiplication of two `unsigned int` values in C guarantees that the result will be correct modulo $M = 2^b$ if b is the word size (the number of bits in X), without having to worry about overflow or the use of multiple precision arithmetic. Many Fortran compilers allow the calling of subroutines written in C.

The randomness properties of LFGs can be improved (without sacrificing too much in speed) by using multiple lags (or "taps") [39, 28, 22, 14, 15, 17], i.e. by combining three or more previous elements of the sequence, rather than two. This type of generator has not yet been widely used or studied, however it seems likely that a 3- or 4-lag additive LFG would be a very fast and effective random number generator.

3.3 Shift Register Generators

Shift register (or Tausworthe) generators [1, 2, 38, 21, 39] are generally used in a form where they can be considered as a special case of a lagged Fibonacci generator using XOR. XOR gives by far the worst randomness properties of any operation for an LFG [1, 2, 15], so these generators are not recommended.

Despite their serious drawbacks, shift register generators have been very popular in the past, mainly because they were comparatively fast. However on modern processors addition is not markedly slower than XOR, so there is little reason to use these generators.

3.4 Combined Generators

Combining two different generators has been shown (both theoretically and empirically) to produce an improved quality generator in many circumstances [2, 32, 6, 42].

Based on an algorithm introduced by Wichmann and Hill [43], L'Ecuyer [32] has shown how to additively combine two different 32-bit LCGs to produce a generator that passes all known statistical tests and has a long period of around 10^{18}, thus overcoming the major drawbacks of standard 32-bit LCGs. This has been implemented in a program known as `RANECU` [32, 5]. Combining two LCGs in this way is effectively a more efficient way of implementing an LCG with a much larger modulus [44]. Recently L'Ecuyer *et al.* [41] have implemented combined 48-bit and 64-bit LCGs and MRGs, with even larger periods and better randomness properties. A combined 32-bit LCG is substantially slower than a standard 32-bit LCG, although it is more appropriate to compare it to a 64-bit LCG (which has the same period), in which case the performance is similar, at least on a 32-bit machine where multiple-precision arithmetic is required for the 64-bit LCG.

Other proposed combined generators include algorithms combining an LFG with an LCG [2], or an LFG with a simple Weyl (or arithmetic sequence) generator, which is the basis for the `RANMAR` generator [5, 45] commonly used in computational physics applications. The addition of the Weyl generator greatly improves the randomness properties over the single additive LFG, but `RANMAR` still fails some Monte Carlo tests [15], since the lag used ($p = 97$) is much too small. If you are using this generator, you should greatly increase the lags, to at least (1279,1063).

4 Parallel Random Number Generators

Many different parallel random number generators have been proposed, but most of them use the same basic concept, which is to parallelize a sequential generator by taking the elements of the sequence of pseudo-random numbers it generates and distributing them among the processors in some way. There are three basic ways to do this:

1. **Leapfrog** – The sequence is partitioned in turn among the processors like a deck of cards dealt to card players.

2. **Sequence splitting** – The sequence is partitioned by splitting it into non-overlapping contiguous sections.

3. **Independent sequences** – For some generators, the initial seeds can be chosen in such a way as to produce long period independent subsequences on each processor.

The first two techniques are closely analogous to the cyclic and block methods for data distribution in a data parallel language [53], while the third technique is loosely analogous to a scattered data distribution [49].

Finding a good parallel random number generator has proven to be a very challenging problem, and is still the subject of much research and debate. One of the reasons good parallel random number generators are so hard to create is that any small correlations that exist in the sequential generator may be amplified by the method used to distribute the sequence among the processors, producing stronger correlations in the subsequences on each processor. Inter-processor correlations may also be introduced. Also, the method used to initialize a parallel random number generator (i.e. to specify the seeds for each processor) is at least as important as the algorithm used for generating the random numbers, since any correlations between the seeds on different processors could produce strong inter-processor correlations.

In this section we describe some of the most common parallel random number generators. More information is available in other review articles [4, 46] and in the references given in this section.

4.1 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. A simple way to achieve this goal 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, since each processor leapfrogs by N in the sequence [47, 48, 49, 50, 4]. In order to use this method we need to be able to easily jump ahead in the sequence. This can be done quite easily for linear congruential generators, and merely involves replacing the multiplier a and the additive constant c by new values $A = a^N$ and $C = c(a^N - 1)/(a - 1)$ (both modulo M) [47, 48, 49, 50]. Jumping ahead in the sequence can also be done for combined LCGs [6, 51] and shift-register generators [52], but is not practical for LFGs using addition or multiplication, since the computations are much more complex, making it too slow for practical use.

A 48-bit LCG using the leapfrog technique has been used on vector machines such as the CRAY and CYBER-205 [47, 4, 8], and for the intrinsic random number generator function used in IBM's XL HPF High Performance Fortran release (which offers the option of a 32-bit or a 48-bit LCG – beware that the totally inadequate 32-bit LCG is the default!) [61].

A potentially serious problem with the leapfrog method for LCGs is that although the multiplier a may be chosen to perform well in the Spectral Test, there is no guarantee that $A = a^N$ will also have good spectral properties, particularly since N will in general be arbitrary. This method could work well in situations for which N is fixed (for example, if an application is always run on the same number of processors), since it would be possible to choose a multiplier so that both a and a^N do well in the Spectral Test. However it is not possible to choose a multiplier for which a^N has good spectral properties for any value of N, so this algorithm is not recommended as a general-purpose generator.

There is another potential problem with this type of generator. As mentioned in Section 3.1, linear congruential generators using a modulus that is a power of 2 are known to have correlations between elements in the sequence that are a power of 2 apart. For many parallel computers the number of physical processors is a power of 2, and this is also often the case for the number of abstract processors, i.e. the size of the arrays used in a simulation. 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 the leapfrog linear congruential algorithm used on the CRAY and CYBER-205, which has a power of 2 modulus, has produced spurious results in some Monte Carlo calculations [8, 11]. This problem can be avoided by using a prime modulus.

For these reasons, we do not recommend the leapfrog method, although it may be adequate for many applications. If you do use this type of generator, at least be sure that the LCG is 48-bit or higher (32-bit or higher for a combined LCG) and preferably has a prime modulus.

4.2 Sequence Splitting

Another method for parallelizing random number generators is to split the sequence into non-overlapping contiguous sections, each generated by a different processor [4, 6, 7]. 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. Alternatively, the length of each section of numbers could be chosen much larger than could possibly be used by any processor. If the length of the sections is L, then processor P would generate the sequence

$$X_{PL}, X_{PL+1}, X_{PL+2}, \ldots$$

This method also requires the ability to quickly jump ahead in the sequence by a given amount. It is therefore restricted primarily to the same type of generators as the leapfrog method, however in this case it is also possible to use additive lagged Fibonacci generators. Although jumping ahead using additive LFGs is too slow to do every time a number is generated (which is required for leapfrog), it may be fast enough to be feasible for sequence splitting, which only needs to be done once, in the initialization of the generator. A parallel generator of this type has been implemented by Brent [7], who presents an initialization method that takes $O(r \log n)$ time to jump ahead n for a lag r. An implementation using sequence splitting of a combined linear congruential generator has been given by L'Ecuyer and Côté [51].

A possible problem with this method is that although the sequences on each processor are disjoint (i.e. there is no overlap), this does not necessarily mean that they are uncorrelated. In fact it is known that linear congruential generators with modulus a power of 2 have long-range correlations that may cause problems, since the sequences on each processor are separated by a fixed number of iterations (L). Other generators may also have subtle long-range correlations that could be amplified by using sequence splitting.

A 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 [53]) 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 physical processors. For a combined LCG, this requires only two integer values per array element to store the state of the generator, which should not be too great a memory requirement. This method appears capable of providing a very good parallel random number generator, particularly for data parallel languages such as HPF [54].

4.3 Independent Sequences

The previous two methods were restricted to generators for which arbitrary elements of the sequence could be quickly and easily computed. This means they are impractical for LFGs using multiplication, which is unfortunate since these are among the best sequential generators available.

There is, however, an even simpler way to parallelize a lagged Fibonacci generator, which is to run the same sequential generator on each processor, but with different initial lag tables (or *seed tables*) [55, 56]. In fact this technique is no different to what is done on a sequential computer, when a simulation needs to be run many times using different random numbers. In that case, the user just chooses different seeds for each run, in order to get different random number streams.

This method is similar to sequence splitting, in that each processor generates a different, contiguous section of the sequence. However in this case the starting point in the sequence is chosen at random for each processor, rather than computed in advance using a regular increment. This has the advantage of avoiding (or at least reducing) possible long-range correlations, but only if the seed tables on each processor are random and independent.

The initialization of the seed tables on each processor is a critical part of this algorithm. Any correlations within the seed tables or between different seed tables could have dire consequences. This leads to the Catch-22 situation of requiring an excellent parallel random number generator in order to provide a good enough initialization routine to implement an excellent parallel random number generator. However this is not as difficult as it seems – the initialization could be done by a combined LCG, or even by a different LFG (using different lags and perhaps a different operation).

A potential disadvantage of this method is that since the initial seeds are chosen at random, there is no guarantee that the sequences generated on different processors will not overlap. However using a large lag eliminates this problem to all practical purposes, since the period of these generators is so enormous that the probability of overlap will be completely negligible (assuming that the initial lag tables are not correlated). In fact, the likelihood of overlap is even less than might be expected, due to a useful property of lagged Fibonacci generators. Any LCG produces a single periodic sequence of numbers, with the different seeds just providing different starting points in the sequence. However, LFGs have many disjoint full-period cycles [31, 63], so two different seed tables may produce two completely different non-overlapping periodic sequences of numbers. In fact, since the number of such disjoint cycles is $2^{(p-1)(b-1)}$ for suitably chosen parameters [31, 63], then the probability of different processors producing the same cycle should be completely negligible (for example, values of $p = 127$ and $b = 32$ give roughly 10^{1000} disjoint cycles!). Of course this assumes that the initial seed tables are really random, and do not have any correlations that might negate this argument.

This type of generator is quite popular, and has been implemented for a number of parallel computers and parallel languages. The Connection Machine Scientific Software Library (CMSSL) routine `FAST_RNG` [57] uses an additive LFG with the seeds being initialized by a different parallel random number generator (`CMF_RANDOM`, described in Section 4.4). The interface to this routine allows the user to specify the lag, so in principle the routine can be very good, although the CMSSL documentation suggests using a lag of 17, which is much too small to ensure adequate randomness properties.

The Maspar (or DECmpp) uses `p_random`, a parallel version of `RANDOM`, the standard Unix LFG `random` [60]. The initial implementation of this generator gave extremely poor quality pseudo-random numbers (the lower order bits were not even uniformly distributed), due to a poor initialization of the seed tables on each processor, which left them highly correlated. This was greatly improved in a later release, although the newer version still failed a Monte Carlo Ising model test [23, 24], presumably because the new method for initializing the generator is still introducing some correlations between the seed tables on each processor.

As with sequence splitting, just because the sequences on each processor do not overlap, does not necessarily mean they are uncorrelated. However in this case, if each processor does indeed generate part of a disjoint full-period cycle, there are some theoretical arguments for why any correlations between processors might be expected to be small [63].

Mascagni *et al.* have proposed a method for initializing the lag tables on each processor which guarantees that each processor will generate a sequence from a different full-period cycle [63]. It has been suggested that this method be established as a standard parallel random number generator [64, 65], and it has been used for the intrinsic random number generator in the Portland Group `PGHPF` High Performance Fortran [62]. The method used for seeding the lag tables is similar in complexity to jumping ahead in the sequence, so the initialization time increases with the size of the largest lag, and the current implementation is restricted to additive LFGs with lags ranging from 17 to 127 [63]. Improved techniques may be needed to make the initialization fast enough to be practical with the much larger lags required for acceptable randomness properties. Note however that although an initialization time of say, one minute, would be unacceptable for a general-purpose random number generator, for specific applications such as large-scale Monte Carlo simulations, which usually require many hours of running time, it would be perfectly adequate.

A deficiency of the independent sequence method is that, like sequence splitting, it does not produce the same sequence for different numbers of processors. However, as with sequence splitting, this can be achieved by assigning a separate generator (i.e. a different lag table) to every abstract, rather than physical, processor. This method is used in the CMSSL `VP_RNG` generator [57], and the `PGHPF` implementation mentioned above. A major problem with this method is that each abstract processor needs to have its own lag table, which becomes an exorbitant memory requirement if the lag is large enough to ensure adequate randomness properties. Both the CMSSL and PGHPF implementations use a lag of 17, which is too small. It might be possible to overcome this problem by using a multiplicative, rather than additive, lagged Fibonacci generator, for which a lag as small as 17 is enough to pass all current empirical tests [2, 15].

Of course, if the user is willing to forgo the luxury of generating the same sequence for any number of processors, then memory is not a problem, and the independent sequence method for large-lag additive or multiplicative LFGs is one of the best methods currently available for generating random numbers on parallel computers, as long as the initialization of the lag tables is done properly.

4.4 Other Methods

The cellular automata generator [58] is a generalization of the shift register generator, based on cellular automata rules. A parallel version called CMF_RANDOM is provided by Thinking Machines [59]. Both the sequential and parallel versions of this generator have passed many of the standard statistical tests [58, 59], however CMF_RANDOM failed a Monte Carlo Ising model test [23, 24] and therefore cannot be recommended. This generator is also much slower than those provided in the CMSSL, so it is not often used for large-scale simulations.

There are a number of other methods used for implementing parallel random number generators, including using a different generator for each processor. These methods are not covered here, since they are not widely used.

5 Conclusions and Recommendations

The main recommendation we would give to someone who needs to use a random number generator on a parallel computer is very simple – never trust a parallel random number generator. In particular, never trust the default random number generator provided with the system you are using. We would offer the same advice for sequential random number generators. Many people (including this author) would have spared themselves a lot of tribulation and saved a lot of time if they had followed this simple tenet in the past.

There are very sound reasons for adopting such a skeptical stance. Developing a good, efficient algorithm for generating pseudo-random numbers on a computer is a very difficult problem, especially for parallel computers. The theoretical understanding of random number generators is rather limited, and no amount of empirical testing can ever determine how well a given generator will perform for a new application. There is a long and inglorious history of various random number generators being proposed, studied, tested, approved, advocated, widely used, and then being found to perform poorly in certain circumstances or for certain applications. Unfortunately, many generators have been (and continue to be) advocated, made available in software libraries, and widely used, even after they have been shown to be flawed.

If a generator is shown to fail a certain empirical test, that does not necessarily mean that it will also perform poorly for your application, or the results you spent many months gathering using that generator are now invalid. However, to avoid this possibility, it is **strongly** recommended that for any computation requiring random numbers, at least two very different random number generators should be used and the results compared, in order to check that the random number generator is not introducing a bias.

On a sequential computer, good generators to use are:

- a multiplicative lagged Fibonacci generator with a lag of at least 127, and preferably 1279 or more;
- a 48-bit or preferably 64-bit linear congruential generator that performs will in the Spectral Test and has a prime modulus;
- a 32-bit (or more) combined linear congruential generator, with well-chosen parameters, such as those recommended by L'Ecuyer;
- if speed is an issue (but note the comment on this in Section 2.1), use an additive lagged Fibonacci generator with a lag of at least 1279 and preferably much greater, possibly combined with another generator, as in RANMAR, or using 3 or more lags.

All of the parallel random number generators covered in this review have some limitations or possible problems. There has also been very little empirical testing done on parallel random number generators, and few theoretical results are known. It is therefore much more difficult to recommend good parallel random number generators, but with that caveat, we will recommend:

- A combined linear congruential generator using sequence splitting;
- A lagged Fibonacci generator, although great care must be exercised in the initialization procedure, to ensure that the seed tables on each processor are random and uncorrelated.

  If your application does not require the same results for different numbers of processors, we recommend using a large-lag multiplicative (or additive) generator with a different (randomly initialized) lag table on each physical processor. This could be combined with another generator, as in RANMAR.

  Otherwise a small lag must be used due to memory constraints, as with the PGIHPF and VP_RNG generators, however we recommend using multiplication rather than addition.

Many of the past problems with random number generators have been caused in part by the rapid pace of improvement in computers. 32-bit LCGs were perfectly adequate for many years, but the speed of modern processors has rendered them obsolete. This has been noted in the documentation for most (but unfortunately not all) implementations of these generators, which are kept only for backward compatibility. Shift register generators became very popular because XOR was so much faster than addition and multiplication on processors available at that time, but that is no longer the case, and their relatively poor randomness properties now far outweigh their slight performance advantage. Early implementations of lagged Fibonacci generators used very small lags, even though larger lags were known to give better results, because of worries about memory constraints that are no longer a problem on current computers (except for data parallel implementations). Many comments on the quality of various generators have been made based on statistical tests performed many years ago, using samples of random numbers that are so small as to be almost irrelevant to the results of simulations done on current (and future) high-performance computers.

The improvement in computer performance continues unabated, of course, and it is crucial that the implementation and testing of random number generators keeps pace with these changes. 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 problems as Monte Carlo simulation of lattice gauge theory and condensed matter physics [66]. Such large-scale Monte Carlo simulations will easily exhaust the period (of roughly 10^{18}) of 64-bit LCGs or 32-bit combined LCGs. It will therefore be necessary in the near future to move to very long period generators such as large-lag LFGs or combined 64-bit LCGs or MRGs (which have periods large enough for a Petaflop-age-of-the-universe computation!).

Since faster computers and better algorithms are improving the precision of computer simulations at a rapid pace, it is important to continue to search for better random number generators, and to make more precise and varied tests of the randomness properties of these generators. This is particularly true for parallel computers, where satisfactory algorithms are still lacking.

5.1 Recommended Random Number Generator Software

The following random number generator software is recommended for parallel computers. Software catalog entries for each of these programs are available at the National HPCC Software Exchange (NHSE).

- Combined linear congruential generators with parameters recommended by L'Ecuyer, parallelized using sequence splitting.
  - `RANECU` from CERNLIB
- Lagged Fibonacci generator using multiplication, parallelized using independent sequences.
  - `FIBMULT` from Syracuse University
- Lagged Fibonacci generator using addition, parallelized using independent sequences. Be sure to use the largest possible lag.
  - Scalable Parallel Random Number Generator (SPRNG) Library from NCSA
  - `FIBADD` from Syracuse University

Acknowledgements

I would like to thank Sung-Hoon Ko, Alan Sokal, Kari Kankaala, John Apostolakis, Alan Middleton, and Enzo Marinari for their help, discussions, and suggestions. 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.

6 Glossary

combined generator
A random number generator that combines results from two or more different generators into a single resultant value.

independent sequences
A method of parallelizing a random number generator by choosing initial seeds for each processor in such a way as to produce long period independent (i.e. non-overlapping) subsequences on each processor.

lags
The distances to the previous elements of the sequence that are used to generate the next element. The largest lag is usually referred to as the lag of the generator.

lag table
The array of prior elements of the sequence that must be stored to produce the next element in a lagged Fibonacci generator.

lagged Fibonacci generator (LFG)
A random number generator that combines previous elements in the sequence to generate the next element, using a simple binary arithmetic operation such as multiplication, addition, subtraction, or exclusive OR.

leapfrog
A method of parallelizing a random number generator by partitioning the sequence of numbers among the processors in a cyclic fashion, like a deck of cards dealt to card players.

linear congruential generator (LCG)
A random number generator that uses a simple linear function of the current element in the sequence to produce the next element.

modulus
The maximum value allowed by a random number generator.

multiplier
The constant used to multiply the current value to get the next value for a linear congruential generator.

multiple recursive generator
A generalization of the linear congruential generator, for which any linear combination of previous elements in the sequence can be used to generate the next element.

period
The length of the cyclic sequences produced by a random number generator.

seed
A number chosen by the user to initialize a random number generator.

seed table
The initial values of the lag table for a lagged Fibonacci generator.

sequence splitting
A method of parallelizing a random number generator by partitioning the sequence of numbers in a block fashion, splitting it into non-overlapping contiguous sections.

shift register generator
A random number generator that uses a simple combination of the bits of previous elements in the sequence to produce the next element.

state
The numbers that are required to be stored in order to implement the iterative procedure used in a random number generator. These are the values that must be stored at the end of each run of the program in order for a subsequent run to start at the same point in the sequence of random numbers.

uniform distribution
The probability of a number falling in a particular interval is proportional only to the size of the interval.

References

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

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

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

[4] S.L. Anderson, Random number generators on vector supercomputers and other advanced architectures, *SIAM Rev.* **32**, 221 (1990).

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

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

[7] R.P. Brent, Uniform random number generators for supercomputers, *Proc. Fifth Australian Supercomputer Conference*, Melbourne, December 1992, p. 95.

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

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

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

[11] 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).

[12] 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).

[13] 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).

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

[15] P.D. Coddington, Analysis of random number generators using Monte Carlo simulation, *Int. J. Mod. Phys.* **C 5**, 547 (1994).

[16] I. Vattulainen, T. Ala-Nissila and K. Kankaala, Physical tests for random numbers in simulations, *Phys. Rev. Lett.* **73**, 2513 (1994).

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

[18] 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.

[19] 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.

[20] H. Gould and J. Tobochnik, *An Introduction to Computer Simulation Methods, Vol. 2*, Addison-Wesley, Reading, Mass., 1988.

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

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

[23] 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).

[24] P.D. Coddington, S.-H. Ko, W.E. Mahoney and J.M. del Rosario, Monte Carlo tests of parallel random number generators, in preparation.

[25] 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.

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

[27] G.A. Marsaglia, Random numbers fall mainly in the planes, *Proc. Nat. Acad. Sci.* **61**, 25 (1968).

[28] R.P. Brent, On the periods of generalized Fibonacci recurrences, *Math. Comp.* **63**, 389 (1994).

[29] N. Zierler and J. Brillhart, On primitive trinomials (mod 2), *Information and Control* **13**, 541 (1968).

[30] J.R. Heringa, H.W.J. Blöte and A. Compagner, Mersenne-exponent degrees for random-number generation, *Int. J. Mod. Phys.* **C 3**, 561 (1992).

[31] G.A. Marsaglia and L.H. Tsay, Matrices and the structure of random number sequences, *Linear Algebra Appl.* **67**, 147 (1985).

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

[33] H. Neiderreiter, Quasi-Monte Carlo methods and pseudo-random numbers, *Bull. Amer. Math. Soc.* **84**, 957 (1978).

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

[35] J. Eichenauer-Herrmann and H. Grothe, A remark on long-range correlations in multiplicative congruential pseudo-random number generators, *Numer. Math.* **56**, 609 (1989).

[36] A. De Matteis and S. Pagnutti, Parallelization of random number generators and long range-correlations, *Numer. Math.* **53**, 595 (1988).

[37] D.H. Lehmer, Mathematical methods in large-scale computing units, *Ann. Comput. Lab. Harvard U.* **26**, 141 (1951).

[38] R.C. Tausworthe, Random numbers generated by linear recurrence modulo two, *Math. Comp.* **19**, 201 (1965).

[39] S.W. Golomb, *Shift Register Sequences*, Holden-Day, San Francisco, 1967.

[40] 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**, 331 (1990).

[41] P. L'Ecuyer, F. Blouin, and R. Couture, A search for good multiple recursive generators, *ACM Trans. on Modeling and Computer Simulation* **3**, 87 (1993).

[42] L.-H. Deng, E.O. George and Y.-C. Chiu, On improving pseudo-random number generators, in *Proc. of the 1991 Winter Simulation Conference*, ed. B.L. Nelson *et al*, p. 1035.

[43] B.A. Wichmann and I.D. Hill, An efficient and portable pseudorandom number generator, *Appl. Statist.* **31**, 188 (1982).

[44] P. L'Ecuyer and S. Tezuka, Structural properties for two classes of combined random number generators, *Math. Comp.* **57**, 735 (1991).

[45] G.A. Marsaglia, Toward a universal random number generator, *Stat. Prob. Lett.* **8**, 35 (1990).

[46] W.F. Eddy, Random number generators for parallel processors, *J. Comp. Appl. Math.* **31**, 63 (1990).

[47] W. Celmaster and K.J.M. Moriarty, A method for vectorized random number generators, *J. Comput. Phys.* **64**, 271 (1986).

[48] K.O. Bowman and M.T. Robinson, Studies of random number generators for parallel processing, in *Proc. 2nd Conference on Hypercube Multiprocessors*, ed. M.T. Heath, SIAM, Philadelphia, 1987, p. 445.

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

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

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

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

[53] C. Koelbel *et al.*, *The High Performance Fortran Handbook*, MIT Press, Cambridge, Mass., 1994.

[54] P.D. Coddington, S.H. Ko, O. Odeyemi and C. Stoner, A random number generator for High Performance Fortran, in preparation.

[55] T.-W. Chiu, Shift-register sequence random number generators on the hypercube concurrent computers, in *Proc. of the 3rd Conference on Hypercube Concurrent Computers and Applications*, ed. G. Fox, (ACM Press, New York, 1988), p. 1421.

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

[57] *CM Scientific Software Library*, Thinking Machines Corporation, Reading, Mass., 1993.

[58] S. Wolfram, Random sequence generation by cellular automata, *Adv. Appl. Math.* **7**, 123 (1986).

[59] *CM Fortran User's Guide*, Thinking Machines Corporation, Reading, Mass., 1994.

[60] See the online manual pages for the Maspar.

[61] *XL High Performance Fortran for AIX Language Reference V1.1*, IBM Corporation, 1996.

[62] *pghpf User's Guide V2.0*, The Portland Group, Inc., Wilsonville, Oregon, October 1995.

[63] D.V. Pryor, S.A. Cuccaro, M. Mascagni and M.L. Robinson, Implementation and usage of a portable and reproducible parallel pseudorandom number generator, in *Proc. of Supercomputing '94*, IEEE, 1994, p. 311.

[64] M. Mascagni and D.H. Bailey, Requirements for a parallel pseudorandom number generator, Supercomputing Research Center technical report, unpublished.

[65] Cherri Pancake *et al.*, Specification of Baseline Development Environment, Section 3, Component BDE-3i, in *Guidelines for Writing System Software and Tools Requirements for Parallel and Clustered Computers.*
Available at `http://www.nero.net/~pancake/SSTguidelines/baseline.html`.

[66] P. Rodgers, *Physics World* (Feb. 1991) p. 13;
S. Aoki *et al.*, *Int. J. Mod. Phys.* **C 2**, 829 (1991);
K. Binder, Large-scale simulations in condensed matter physics – the need for a teraflop computer, *Int. J. Mod. Phys.* **C 3**, 565 (1992).