PARALLEL RANDOM NUMBER GENERATORS

Gülin Kaşıkara

Department of Computer Engineering
Middle East Technical University
Fall 2003

This Page Intentionally Left Blank

Abstract

Random number generators (RNG) are used in many applications. For many application, it is important that the generators have good randomness properties. Good random number generators are hard to find and many widely used techniques are shown to be inadequate. Finding high-quality, efficient algorithms for random number generation on parallel computers is even more difficult. This report covers the theory behind random number generation. The properties of parallel and sequential random number generators and requirements for them are defined by comparison. The question of why do we need random number generators are explained. Some parallel and serial random number generation algorithms are covered in detail emphasizing the facts that lead to the parallelization of random number generators. The advantages and disadvantages of both parallel and serial algorithms are discussed. Some recommendations for using random number generators on parallel computers are provided.

Table of Contents

1 Introduction 1

2 Sequential Random Number Generation 2

2.1 Requirements for Sequential Random Number Generators 2

2.2 Sequential Random Number Generators 3

2.2.1 Linear Congruential Generators (LCG) 3

2.2.2 Lagged-Fibonacci Generators (LFG) 3

2.2.3 Combined Generators 4

3 Parallel Random Number Generation 5

3.1 Requirements for Parallel Random Number Generators 5

3.2 Types of Parallel Random Number Generators 5

3.3 Distributed Parallel Random Number Generators 6

3.3.1 Splitting Techniques 6

3.3.2 Parameterizing Techniques 10

4 Parallel RNGs vs Serial RNGs 12

5 Conclusion 13

6 References 14

Appendices

Table 1 Properties of serial random number generators 15

Table 2 Properties of parallel random number generator 16

1 Introduction

Random numbers are widely used for simulations in computational science and engineering such as, in Monte Carlo method to estimate a many dimensional integral by sampling the integrand, in modeling random processes in nature such as those arising in ecology or economics, in cryptography, where randomness is used to hide information from others. Random numbers may also be used in games, for example during interaction with the user [4].

Random numbers are in practice computed using deterministic algorithms. So, these are more accurately called pseudo-random number generators. In some simulations, the quality of pseudo-random numbers is not that important. However, in many problems for which random numbers are most heavily used, the quality of the random number generator is crucial.

The aim of this article is to explain the theory behind the random number generation and give basic information about both serial and parallel random number generators. In order to understand parallel random number generators, sequential random number generators (SRNG) must be fully understood. Firstly, the properties of sequential random number generators and the requirements for SRNG are covered. The questions of which applications are using SRNGs and why do we need random number generators are answered. Some examples of SRNGs are given like linear congruential generators, lagged-Fibonacci generators and combined generators to understand the theoretical background behind the subject. Lastly, the deficiencies of sequential random number generators are shown and the facts that lead us to the parallel random number generators (PRNG) are explained.

After having a general view of SRNGs, the properties of PRNGs and the requirements for them are covered. Types of PRNGs are explained like centralized parallel random number generators, replicated random number generators and distributed random number generators. The emphasis is given on the distributed random number generators. Two main techniques of distributed random number generators are explained (splitting and parameterization). Splitting technique is based on splitting the full-period sequences of conventional pseudorandom number generators into subsequences. Given the subsequences, each parallel process requiring random numbers is then given a different subsequence for this purpose. The splitting methods considered include breaking the original sequence into non-overlapping subsequence blocks, using leapfrog technique to make subsequences, using the Lehmer tree method (Random tree method), and using recursive-halving leap ahead for subsequence definition. After discussing splitting technique in detail, another alternative is presented, using parameterized, full period pseudorandom number sequences, and several methods based on parameterization are discussed. Parameterized versions of commonly used pseudorandom number generators are described like linear congruential generators, shift register generators and lagged-Fibonacci generators.

After gaining the knowledge behind the parallel and sequential random number generators, what is gained and what is lost by parallelization is shown and the advantages and disadvantages of PRNGs in contrast to sequential ones are made clear.

2 Sequential Random Number Generation

RNGs are used for generating an array of numbers that have a random distribution. Generation is done by via a function called generator, which is defined as, when applied to a number, yields the next number in the sequence. The RNGs used in practice do not actually generate numbers that are truly random. These programs use deterministic algorithms and are actually pseudo-random number generators. The resulting sequence looks statistically independent and uniformly distributed.

RNGs require the user to specify an initial value, or seed. Initializing the generator with the same seed will give the same sequence of random numbers. If different sequences are needed, different seeds must be used .

Many widely used RNGs have been shown to have quite poor randomness properties that lead to incorrect results in certain applications. It is better to use a RNG that has been thoroughly tested and recommended. For applications in which random numbers are only used occasionally, the quality of the generator will probably not matter, however in applications which use a lot of random numbers, such as Monte Carlo simulations, the quality of the generator is crucial and poor generators can produce incorrect results [1].

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. have long period
6. are reproducible (for debugging purposes),
7. are fast
8. are portable (the same on any computer),
9. can be changed by adjusting an initial seed value,
10. can easily be split into many independent subsequences,
11. can be generated rapidly using limited computer memory.

In practice it is impossible to satisfy all these requirements exactly. For practical purposes, it is required 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.

It is important to realize that the numbers generated by random number generators are not random, but are instead pseudo-random. Since random number generators use deterministic algorithms, they cannot be completely uncorrelated, and since the numbers they produce are reproducible, they cannot be truly random. However, many applications that use random numbers require that the random number generator be of highest quality.

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. It is logical to sacrifice a

little speed for much better randomness properties. In using a random number generator, it is usually better to be slow than sorry [1].

2.2 Sequential Random Number Generators

In this section, the most popular and commonly used random number generators are introduced.

2.2.1 Linear Congruential Generators (LCG)

The most commonly used random number generator is the LCG. It is based on the following iterative scheme:

$X_n = (a X_{n-1} + c) \bmod m$

where m is the modulus, a the multiplier, and c the additive constant that may be set to 0. (a, c, m) must be chosen carefully for a long period, good uniformity and randomness properties. The size of the modulus constrains the period, and is usually a prime or a power of 2.

LCGs work very well for most applications but are known to have some major defects. The main problem is the least significant bits of the numbers produced are correlated. Another problem is that many commonly used LCGs use a modulus m that is a power of 2 since it is fast and convenient to implement on a computer. However, this approach produces highly correlated low order bits and long-range correlations for intervals that are a power of 2. To avoid these problems, it is best to use modulus that is prime rather than a power of 2. Moreover, increasing the precision of the generators for example, by using 64-bit numbers rather 32-bit ones can decrease the effects of these regularities. However, there are practical limits to this approach. If modulus becomes greater than machine precision, then much slower multi precision arithmetic must be used, so in practice precision should not be made larger than the machine precision [1].

Despite these known problems, large precision LCGs with well-chosen parameters appear to work very well on sequential computers. Besides, LCGs are the best choice of generators for applications that don't expect the lower-order bits to be random [1].

Multiple recursive generators (MRG) generalize LCGs by using a recurrence of the form,

$X_n = (a_1 X_{n-1} + a_2 X_{n-2} + a_3 X_{n-3} + \ldots\ldots + a_k X_{n-k} + b) \bmod m$

For a given value k. Choosing $k > 1$ will increase time taken to generate each number, but will greatly improve the period and randomness properties of the generator.

2.2.2 Lagged-Fibonacci Generators (LFG)

The name of the generator comes from the Fibonacci sequence. LFGs generate random numbers from the following iterative scheme:

$X_n = X_{n-p} \Theta X_{n-q} \pmod m$

where p and q are the lags, satisfying the conditions $p > q > 0$ and Θ is any binary arithmetic operation (addition, multiplication or XOR). The current X is determined by the values of X from p and q places ago. For an LFG, p initial values $X_0, X_1,.....,X_{p-1}$ are needed. This method requires storing the p previous values in the sequence in an array called a lag table.

As in LCGs, the parameters p, q and m must be carefully chosen to provide good randomness properties and the largest period. An advantage of this generator is that the period can be made arbitrarily large by just increasing the lag p. This also improves randomness properties since smaller lags mean higher correlations between the numbers in the sequence. One problem with LFG is that p words of memory must be kept current, whereas LCG requires only that the last value of X be saved.

The randomness properties of LFGs are best when multiplication is used, with addition being next best, XOR being by far the worst [1].

LFGs using addition 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 [1]. Each pseudo-random number can be generated with a single floating-point addition and a modulus operation. Great care must be taken when choosing the lags for additive LFGs. Usually, much too small lags to give adequate randomness properties are chosen in many applications. Increasing the lag improves the randomness properties of the generator. A lag greater than 1000 is recommended for an additive LFG [1].

The choice of the lag may affect the speed of the generator, depending on the computer used. If a vector processor is used, a larger lag may improve performance, since the vector lengths are larger. If a scalar processor with limited cache is used, having a large lag may reduce the performance.

Multiplicative LFGs have seen little use. Although, slower than additive LFGs, they are as fast as 32-bit LCGs. They can be used with smaller lags than additive LFGs. One of the problems of multiplicative LFGs is handling the possible overflow of multiplication [1].

Multiplicative LFGs are rarely used but have longer periods and good randomness properties. Where as, additive LFGs are popular since they are simple and easy to implement [1].

The randomness properties can be improved by using multiple lags by combining three or more previous elements of the sequence, rather than two.

2.2.3 Combined Generators

In order to improve the quality of a generator, two different generators arre combined. Combined generators have the following recurrence relation,

$S_n = (X_n + Y_n) \bmod m$

Additively combining to different 32-bit LCGs is more efficient than implementing an LCG with a much larger modulus [1].

3 Parallel Random Number Generation

The goal of parallel random number generation is to design a random number generator that produces random sequences of integers on each processor in a parallel computing environment [7].

Driving force to parallelize sequential random number generators comes from the necessity to get higher speed in Monte Carlo applications. In order to get higher speed, these applications make extensive use of the parallel computers, since these calculations are particularly well suited to such architectures and often require very long run [4].

In this section, firstly, requirements for parallel random number generators are discussed. Then, some methods for generating random number distributedly are given. These methods all assume a good source of sequential random numbers which is transformed in some manner to a sequence of normally distributed random numbers.

3.1 Requirements for Parallel Random Number Generators

In addition to the requirements for an ideal SRNG, 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.
3. There should be no correlations between the sequences on different processors (inter-processor correlation).
4. The same sequence of random numbers should be produced for different numbers of processors, and for the special case of a single processor [7].
5. The speed of generation of the numbers on each processor and the amount of memory required per processor should be independent of the number of processors.
6. 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. Among the above-mentioned requirements of PRNGs, the most important one is the requirement that there should be no inter-processor correlation. This issue did not arise in the case of serial random number generators.

3.2 Types of Parallel Random Number Generators

There are three general approaches to the generation of random numbers on parallel computers: centralized, replicated, and distributed. In the centralized approach, a sequential generator is encapsulated in a task from which other tasks request random numbers. This avoids the problem of generating multiple independent random sequences, but is unlikely to provide good performance. Furthermore, it makes reproducibility hard to achieve: the response to a request depends on when it arrives at the generator, and hence the result computed by a program can vary from one run to the next.

In the replicated approach, multiple instances of the same generator are created. Each generator uses either the same seed or a unique seed, derived, for example, from a task identifier. Clearly, sequences generated in this fashion are not guaranteed to be independent and, indeed, can suffer from serious correlation problems. However, the approach has the advantages of efficiency and ease of implementation and should be used when appropriate.

In the distributed approach, responsibility for generating a single sequence is partitioned among many generators, which can then be parceled out to different tasks. The generators are all derived from a single generator; hence, the analysis of the statistical properties of the distributed generator is simplified. Focus will be on the distributed generators in the next section.

3.3 Distributed Parallel Random Number Generators

There are several methods for creating distributed parallel random number generators. These methods can be grouped under two basic techniques, which are splitting and parameterization.

The underlying idea behind the splitting technique 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. On the other hand, parameterizing technique identifies a parameter in the underlying recursion of a serial random number generator that can be varied. Each valid value of this parameter leads to a recursion that produces a unique, full-period stream of random numbers [3,5].

Finding a good parallel random number generator is a very difficult problem [2]. One of the reasons 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-process correlations may also be introduced. Also, the method used to initialize a parallel random number generator is at least as important as the algorithm used for generating the random numbers, since any correlation between the seeds on different processorscould produce strong inter-processor correlations.

3.3.1 Splitting Techniques

Splitting techniques based on the following concept, that 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 several methods to do this, which differs slightly but have the same basic concept, like sequence splitting, leapfrog, independent sequences, and random tree method. A top-down approach can be taken to choose a splitting scheme and a SRNG. There are five properties that make a SRNG suitable for splitting. These are:

1. existence of a fast-leap-ahead algorithm,
2. period long enough to be split,
3. serial pseudo randomness,
4. substream independence,
5. fast serial implementation [5].

By considering these five important factors, different splitting techniques are discussed one by one.

3.3.1.1 Sequence Splitting (Blocking Technique)

Another method for parallelizing random number generators is to split a serial random number sequence into non-overlapping contiguous sections, each generated by different processors. If there are N processors, and the period of the serial sequence is P, then the first processor gets the first P/N random numbers, the second processor gets the second P/N random numbers, etc. This method required a fast way to advance the serial sequence P/N steps. It turned out that LCG was a good candidate for this, but also, it is possible to use additive LFGs since jumping ahead is done only once in the initialization of the generator.

A possible problem with this method is that although the sequences on each processor are disjoint, this does not necessarily mean that they are uncorrelated. In fact, it is known that LCG with power of modulus, causes long-range correlations, which could become short-range inter-stream or inter-processor correlations in parallel generators.

One danger of this method is that if the user happens to consume more random numbers than expected, then the sequences could overlap. Another disadvantage of this kind of generator is that it does not produce the same sequence for different number of processors [1, 3, 5].

3.3.1.2 The Leapfrog Method

In this approach, the sequence of a serial generator is partitioned in turn among multiple processors like a deck of cards dealt to card players. If there are N processors, each processor leapfrogs by N in the sequence. For example, processor i gets X_i , X_{i+N} , X_{i+2N} , etc. To produce the same sequence of random numbers for different number of processors, this method can be used. In order to use this method, jumping ahead in the sequence should be done easily. This can be done quite easily with LCGs but not practical for LFGs.

This method has serious problems that long-range correlations in the original sequence can become short-range inter-stream correlations in the parallel generator. Since, it is known that LCGs using a modulus that is a power of two have correlations between elements in the sequence that are a power of two apart. Moreover, for many parallel computers, the physical number of processors is a power of two. So, this method becomes useless.

It may be adequate for some applications. If it is to be used, number of processors must be fixed and the modulus must be prime [1, 3, 5].

3.3.1.3 Independent Sequences

The previous two methods are impractical for LFGs using multiplication. There is 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. This method is similar to sequence splitting, in that each processor generates a different contiguous section of the sequence. 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 reducing possible long-range correlations, but only if the seed tables on each processor are random and independent [3].

The initialization of the seed tables on each processor is a critical part of this algorithm. This leads to requiring an excellent parallel random number generator in order to provide a good enough initialization routine to implement an excellent parallel random number generator. The initialization can be done by a combined LCG, or even by a different LFG.

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. 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 different seeds just providing different starting points in the sequence.

As with sequence splitting, just because the sequences on each processor do not overlap, does not necessarily mean they are uncorrelated. A deficiency of the independent sequence method is that, like sequence splitting it does not produce the same sequence for different number of processors.

3.3.1.4 The Random Tree Method (Lehmer Tree Method)

The idea of random tree method is to choose two linear congruential generators, called the right and the left generators, L and R, that differ only in the values used for a, to construct a tree of pseudorandom numbers.

$L_{k+1} = a_L L_k \bmod m$

$R_{k+1} = a_R R_k \bmod m$

Left generator L is applied to a seed L_0 and a random sequence is generated. Application of the right generator R to the same seed generates a different sequence. By applying the right generator to elements of the left generator's sequence (or vice versa), a tree of random numbers can be generated. By convention, the right generator R is used to generate random values for use in computation, while the left generator L is applied to values computed by R to obtain the starting points R^0, R^1, etc., for new right sequences. In other words, left generator leaps ahead in the right sequence to seed the next right generators. Thus, for linear congruential generators, the concept of Random tree can be thought of as using a precomputed leap-ahead value to split the sequence into a fixed number of sequences.

The strength of the random tree method is that it can be used to generate new random sequences in a reproducible and noncentralized fashion. This is valuable, for example, in applications in which new tasks and hence new random generators must be created dynamically. Before creating a new task, a parent task uses the left generator to construct a new right generator, which it passes to its new offspring. The new task uses this right generator to generate the random numbers required for its computation. If it in turn must generate a new task, it can apply the left generator to its latest random value to obtain a new random seed.

A deficiency of the random tree method is that there is no guarantee that different right sequences will not overlap. The period of R is usually chosen to be near to m, because this maximizes the quality of the random numbers obtained by using the generator. Hence, the starting points R^i returned by the left generator are likely to be different points in the same

sequence, in which case we can think of L as selecting random starting points in the sequence constructed by R. If two starting points happen to be close to each other, the two right sequences that are generated will be highly correlated.

In some circumstances, it can be known that a program requires a fixed number of generators. In this case, a variant of the random tree method called the leapfrog method can be used to generate sequences that can be guaranteed not to overlap for a certain period.

Let n be the number of sequences required. Then a_L and a_R is defined as a and a^n respectively.

$L_{k+1} = a\ L_k \bmod m$

$R_{k+1} = a^n\ R_k \bmod m$

Then, n different right generators ($R^0 \ldots R^{n-1}$) are created by taking the first n elements of L as their starting values. The name leapfrog method refers to the fact that the i th sequence R^i consists of L^i and every n th subsequent element of the sequence generated by L (Figure 3.3.1.4.1)**.** As this method partitions the elements of L, each subsequence has a period of at least P/n, where P is the period of L. In addition, the n subsequences are disjoint for their first P/n elements.

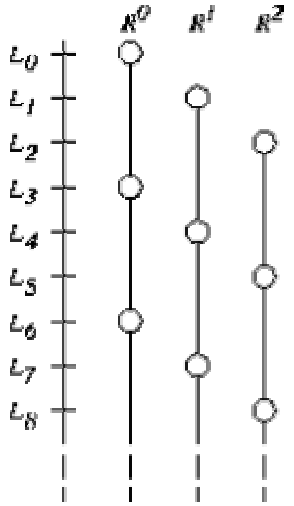

Figure 3.3.1.4.1 The leapfrog method with n=3. Each of the three right generators selects a disjoint subsequence of the sequence constructed by the left generator's sequence.

In other situations, the maximum number, n, of random values needed in a subsequence is known but not the number of subsequences required. In this case, a variant of the leapfrog method (modified leapfrog) can be used in which the role of L and R are reversed so that the elements of subsequence *i* are the contiguous elements (Figure 3.3.1.4.2) as follows:

$L_{k+1} = a^n\ L_k \bmod m$

$R_{k+1} = a\ R_k \bmod m$

It is not a good idea to choose n as a power of two, as this can lead to serious long-term correlations.

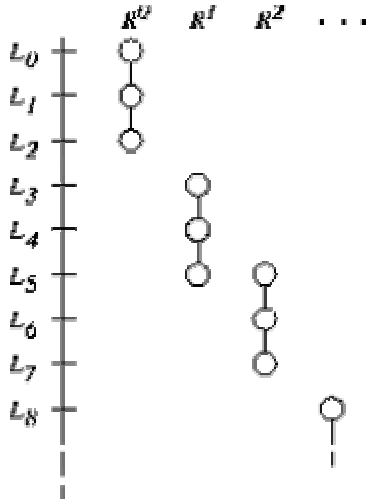

Figure 3.3.1.4.2 Modified leapfrog with n=3. Each subsequence contains three contiguous numbers from the main sequence.

3.3.1.5 Recursive Halving Leap – Ahead

Recursive halving leap ahead is the extension of the Random tree concept to more general splittings. An undesirable property of the Random tree is that the left generator is fixed. This means that splitting a previously split sequence can lead to considerable overlap with the grandparent sequence. To overcome this problem, variable sized leap-ahead values are used. Thus the first split leaps ahead about one half of the period, the next about one-quarter, and so on. This procedure is called recursive halving leap-ahead [3].

3.3.2 Parameterizing Techniques

The parameterization method is one of the latest methods of generating parallel random numbers. The exact meaning of parameterization depends on the type of parallel random number generator. This method identifies a parameter in the underlying recursion of a serial random number generator that can be varied. Each valid value of this parameter leads to a recursion that produces a unique, full-period stream of pseudorandom numbers [3,5]. Each processor is given the same PRNG but a different set of parameter values. Hence, each processor executes the same general algorithm, and the same piece of code, only the parameters passed in initialization is varied from process to process [6]. Two different methods for parameterization are given in the next sections.

3.3.2.1 Cycle Parameterization

This method makes use of the fact that some parallel random number generators have more than one cycle. If the seeds are chosen carefully, then it can be assumed that each random sequence starts out in a different cycle so two sequences will not overlap. Thus the seeds are parameterized that is, sequence I gets a seed from cycle I, the sequence number being the parameter that determines the cycle [4]. In other words, consider a single PRNG that has full-period cycles that fall into different equivalence classes depending on the initial seed. This PRNG is then seeded appropriately to ensure that each parallel process uses a different equivalence class [6]. This is the case for the additive lagged-Fibonacci generator, which is parameterized through its initial values.

3.3.2.2 Parameterized Iteration

Just as the disjoint cycles parameterized, many of the iteration functions can be parameterized. This is the case in the parameterization of linear congruential generators and shift register generators. The most important parameter of an LCG is the modulus m. Its size constraints the period, and for implementation reasons it is always chosen to be either prime or a power of two. The parameterization method used is based on the type of modulus that has been chosen. When m is prime, a method based on used of the multiplier a as the parameter has been proposed. An alternative way to use LCGs to make parallel random number generators is to parameterize the additive constant when the modulus is a power of two [3,4,5].

4 Parallel RNGs vs Serial RNGs

As can be seen from Table 1 and Table 2 in appendices, there are several methods for implementing serial and parallel random number generators. Each parallel random number generator no matter which technique is used relies on an underlying sequential random number generator. This fact makes parallel random number generators more prone to errors than serial ones because 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. Besides, inter-process correlations may also be introduced which is not considered in the sequential case. Also, the method used to initialize a parallel random number generator is at least as important as the algorithm used for generating the random numbers, since any correlation between the seeds on different processors could produce strong inter-processor correlations. As a result of these facts, one can easlily understand that finding a good parallel random number generator is a very difficult problem. Moreover, it is realized that there is no such thing as a parallel random number generator that behaves flawlessly for every application [1].

On a sequential computer good generators to use are a 48-bit or 64-bit LCG, a multiplicative LFG with lag greater than 127, or a 32-bit or more combined LCG. If speed is an issue it is recommended to use additive LFG with lag greater than 1279, possibly combined with another generator, or using multiple lags [1].

All of the parallel random number generators have possible problems. Splitting technique suffers from serious problems as can be seen from Table 4.2. When considering a single sequence for parallel use, one must keep in mind that the parent sequence must have a period of enough length to provide all the random numbers required for all the parallel processes. Thus the period of parent generators will have to be very big. Unfortunately, for most generators a longer period implies that the cost of generating each random number goes up. In addition, there is often correlation in the parent sequence. Thus, splitting is not an acceptable way to parallelize random number generators, except with certain types of generators. Hence, parameterization technique can be a better way to parallelize random number generators [3].

5 Conclusion

In this report, it is aimed to give theoretical information about random number generators. Firstly, sequential random number generators are explained in detail. Requirements and methods are given. Then, parallelization of random number generators is discussed. Finally, sequential and parallel random number generators are compared and advantages and disadvantages of each method are determined.

There are several sequential random number generators available. The most important conclusion regarding sequential generators is that good ones may exist, but are hard to find [2]. Linear congruential generators with a short period are certainly to be avoided. Lagged-Fibonacci generators using the XOR operation are also to be avoided; other lagged-Fibonacci generators may be satisfactory if the lags are large. If the operation to be used is addition then the lags should probably be at least 1000 [1].

There are several methods to parallelize random number generators each having its own problems. The driving force is to get higher speed. But, the desire for reproducibility, when combined with speed, is also an important factor and limits the feasible parallelization schemes. Unfortunately, there is not a parallel random number generator that behaves as desired for every application [1].

6 References

[1] P. D. Coddington. Random number generators for parallel computers, 28 April 1997. http://www.npac.syr.edu/users/paulc/papers/NHSEreview1.1/PRNGreview.ps.

[2] P. Hellekalek. Good random number generators are (not so) easy to find, 1998.

[3] M. Mascagni. Theory and software for parallel random number generation.

[4] A. Srinivasan, D. M. Ceperley, and M. Mascagni. Random number generators for parallel applications. In D. Ferguson, J. I. Siepmann, and D. G. Truhlar, editors, *Monte Carlo Methods in Chemical Physics*, Advances in Chemical Physic series, volume 105, New York, 1998. John Wiley and Sons.

[5] M. Mascagni. Parallel pseudorandom number generation. SIAM News, volume 32, number 5.

[6] M. Mascagni, S. A. Cuccaro, M. L. Robinson and D. V. Pryor. Recent developments in parallel pseudorandom number generation. In D. E. Keyes, M.R. Leuze, L. R. Petzold, and D. A. Reed, editors, Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, volume II, pages 524-529, Philadelphia, Pennsylvania, 1993. SIAM.

[7] S. Aluru, G. M. Prabhu, and J. Gustafson. A random number generator for parallel computers, 1992. Parallel Computing volume 18, pages 839-847.

Appendices

SRNG	Formula (mod m)	Advantages	Disadvantages	Recommended Usage
Linear Congruential	$X_n = a X_{n-1} + c$	Last value of X must be kept	Correlation on the least significant bits Long range correlations when modulus is power of two	▪ Use large precision (48 bit or 64 bit) ▪ Use prime modulus
Lagged Fibonacci	$X_n = X_{n-p} \Theta X_{n-q}$	Increase in period by increase in p	p words of memory must be kept	Multiple lags
Additive	$X_n = X_{n-p} + X_{n-q}$	▪ Fast and easy to implement ▪ Floating point computation	Poor randomness than MLFG	Use lag > 1000
Multiplicative	$X_n = X_{n-p} * X_{n-q}$	▪ Good randomness properties ▪ Longer period than additive	▪ Overflow of multiplication ▪ Can use smaller lags than ALFG ▪ Slower than ALFG faster than LCG	Use lag > 127
XOR	$X_n = X_{n-p} \oplus X_{n-q}$		Poor randomness than ALFG	Not recommended
Combined	$S_n = X_n + Y_n$	Better than one generator with larger m		32 bit or more precision LCG

Table 1 Properties of serial random number generators

PRNG	SRNG	Advantages	Disadvantages
Splitting Technique			
Sequence Splitting (Blocking)	LCG ALFG	Disjoint sequences	▪ Requires a fast way to leap ahead ▪ Sequences can be correlated ▪ Short range inter processor correlations ▪ Short range inter stream correlations ▪ Same sequence is not produced for different number of processors ▪ Sequences can overlap when more numbers are required
Leapfrog	LCG	Same sequence is produced for different # of processors	▪ Requires a fast way to leap ahead ▪ Sequences can be correlated ▪ Short range inter processor correlations ▪ Short range inter stream correlations
Random Tree	CLCG + LFG LFG + LFG	▪ Disjoint sequences like Sequence splitting ▪ Reduces possible long range correlations ▪ Reproducible sequences ▪ Better than Sequence splitting and leapfrog	▪ Seed tables must be random and independent ▪ The initialization of the seed tables is critical ▪ Seed tables can overlap ▪ Same sequence is not produced for different number of processors ▪ Different right sequences can overlap.
Recursive Halving Leap- Ahead	LCG + LFG LFG + LFG	Leap-ahead sizes are variable Better than Random tree	
Parameterization Technique			
Cycle Parameterization	ALFG		
Parameterized Iteration	LCG: Prime Modulus, Power of Two SRG		

Table 2 Properties of parallel random number generator