On Parallel Pseudo-Random Number Generation

Chih Jeng Kenneth Tan

School of Computer Science
The Queen's University of Belfast
Belfast BT7 1NN
Northern Ireland
United Kingdom
cjtan@acm.org

Abstract. Parallel computing has been touted as the pinnacle of high performance digital computing by many. However, many problems remain intractable using deterministic algorithms. Randomized algorithms which are, in some cases, less efficient than their deterministic counterpart for smaller problem sizes, can overturn the intractability of various large scale problems. These algorithms, however, require a source of randomness. Pseudo-random number generators were created for many of these purposes. When performing computations on parallel machines, an additional criterion for randomized algorithms to be worthwhile is the availability of a parallel pseudo-random number generator. This paper presents an efficient algorithm for parallel pseudo-random number generation.

Keywords: Randomized computations, Monte Carlo method, Stochastic methods, Pseudo-random number generators, Parallel computing

1 Introduction

Parallel computing has been touted as the pinnacle of high performance digital computing by many. However, many problems remain intractable using deterministic algorithms even on large parallel digital machines. Randomized algorithms, which are in some cases less efficient than their deterministic counterpart, especially when the problem sizes are relatively small, can overturn the intractability of various large scale problems. In the area of computational finance, for example, stochastic algorithms have been crucial for the solution of various problems. Similar examples may be drawn from areas ranging from computational linear algebra, to computational physics, to environmental modeling.

At the heart of the stochastic algorithms, lie a source of randomness. The question of what can be considered random has often been asked. Various physical sources of randomness have been suggested as sources of randomness for randomized algorithms. Not only that such sources are often not repeatable, making it very hard to verify or debug a program written, it has been shown that some of these physical sources are not "sufficiently random". Pseudo-random number generators are often designed to be a source of randomness which can be used in stochastic computations.

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2073, pp. 589–596, 2001.

This paper presents an efficient algorithm for parallel pseudo-random number generator called `PLFG`, which is based on lagged Fibonacci generator algorithm.

2 Pseudo-Random Number Generators

Pseudo-random number generators have been an interest of researchers, since the early days of computing. Putting aside the philosophical issues involved in the question of what is, or can be, considered random, pseudo-random number generators have to cater for repeatable simulations, have relatively small storage space requirements, and have good randomness properties within the sequence generated.

When performing computations on parallel machines, an additional criterion that needs to be satisfied is the availability of a parallel pseudo-random number generator. The streams of pseudo-random numbers used by each processor have to be independent.

In addition, computational requirements such as coding, initialization time, running time, memory footprint, portability and efficiency have to be taken into consideration as well, when designing pseudo-random number generating algorithms [7].

The pseudo-random number generators used today generally fall into the following categories: linear congruential generators, non-linear congruential generators, lagged Fibonacci generators, Tausworthe generators and mixed generators. Regardless of which pseudo-random number generator is used, its algorithm inflates an input of a short number of bits into a much longer sequence of random bits.

2.1 Lagged Fibonacci Generators

Generally, lagged Fibonacci generators are of the form

$$x_i = (x_{i-p_1} \odot x_{i-p_2}) \bmod M$$

where x_i is the next output pseudo-random number, and $\odot$ is a binary operation. The lag values are p_1 and $p_2, p_1 > p_2$. The operations addition (or subtraction), multiplication or bitwise exclusive OR (XOR) are commonly used in place of $\odot$. M is typically a large integer value or 1 if x_i is a floating point number. When XOR operation is used, mod M is dropped. It is obvious that LFGs require a lag table of length p_1 to store $x_j, j = i-1, i-2, \ldots, i-p_1$.

Although multiple lagged Fibonacci generators using XOR operations can be combined to provide good quality sequences [4], the individual sequence obtained by using a lagged Fibonacci generator with XOR operations give the worst pseudo-random numbers, in terms of their randomness properties [5,1,11]. Multiplicative lagged Fibonacci generators have been shown to have superior properties compared to additive lagged Fibonacci generators [5]. Because multiplication operations are still being perceived as being slower than addition or

subtraction operations, additive lagged Fibonacci generators have found more common use than their multiplicative counterpart. However, tests comparing operation execution times have shown that, with current processors and compilers, multiplication, addition and subtraction operations are of similar speeds. Thus, the argument favoring additive operations over multiplicative operations is nulled and multiplicative lagged Fibonacci generators should be preferred.

Care should be taken when choosing the parameters p_1, p_2 and M in order to obtain a long period and good randomness properties. The value of $p_1 > 1279$ was suggested in [2]. Having a large p_1 also improves randomness since smaller lags lead to higher correlation between numbers in the sequence [5,1,2]. In lagged Fibonacci generators, the key purpose of M is to ensure that the output is bounded within the range of the data type.

Initializing the lag table of the lagged Fibonacci generator is also of critical importance. The initial values have to be independent. To obtain these values, another pseudo-random number generator is often used.

With $M = 2^b$, where b is the total number of bits in the data type, additive lagged Fibonacci generator have a maximal period $\Pi_{ALFG} = 2^{b-1}\,(2^{p_1} - 1)$. The maximal period of multiplicative lagged Fibonacci generator, however, is shorter than that of additive lagged Fibonacci generator: $\Pi_{MLFG} = 2^{b-3}\,(2^{p_1} - 1)$. It is obvious that this shorter period should not pose a problem for multiplicative lagged Fibonacci generators if p_1 is large.

If $x_{i-p_1}, x_{i-p_2}, \ldots, x_{i-p_n}$ are used to generate x_i, it is said to be an "n-tap LFG". Empirical tests have shown that $n > 2$ may increase the randomness quality of the sequence generated. In an n-tap additive lagged Fibonacci generator, M can be chosen to be the largest prime $< 2^b$ [3]. It can be proven using the theory of finite fields that pseudo-random numbers generated in such a manner will be a good source of random numbers. For a complete discussion, see [3].

3 Parallelizing Schemes

Several pseudo-random number generator parallelizing schemes exist. The well known ones being leap frog, sequence splitting, independent sequences and shuffling leap frog [2,12]. All these techniques, except the method of independent sequences, require arbitrary elements of the sequence to be generated efficiently. While it is technically possible to parallelize lagged Fibonacci generators with techniques like leap frog, sequence splitting and shuffling leap frog, the amount of inter-processor communication that would be required makes it impractical to parallelize lagged Fibonacci generators in such a manner. But lagged Fibonacci generators can be parallelized very easily with the independent streams method, and may be very efficient. Pseudo-random number generator parallelization by independent streams is also a recommended technique [8]. The independent sequences are obtained by having multiple generators running on multiple processors, but seeded independently. It should be stressed that seeding the lag tables have to be done with care, to ensure independence between the individual lag tables.

When a parallel pseudo-random number generator with independent sequences is used for Monte Carlo simulations, it is analogous to running the simulation multiple times, each time with a different pseudo-random number generator. This is highly desirable in Monte Carlo simulations since the variance can be reduced by $\mathcal{O}(\sqrt{n})$ if n independent trials are being carried out.

4 Parallel Implementation

For the reasons of the superiority of lagged Fibonacci generators, and multiplicative lagged Fibonacci generator in particular, as discussed in [9], a multiplicative lagged Fibonacci generator algorithm was used as a basis for the pseudo-random number generator implemented. An independent sequences scheme was used for parallelization.

The length of the bit table of a pseudo-random number generator determines the number of parameters needed for its initialization. In the case of an linear congruential generator, where only one past value is used to produce the next output, the size of the bit table is the size of the data type used, and only one value is needed for seeding [3]. For a lagged Fibonacci generator however, the bit table consists of the bits of all $x_i, i = 1, 2, \ldots, p_1$ in the lag table. This feature of the lagged Fibonacci generator may have a positive influence on the quality of the pseudo-random number sequence generated.

A parallel pseudo-random number generator has to generate sequences which are independent of each other. This translates to independence between all the bit tables in the parallel pseudo-random number generator, at any point in time. In parallelizing a pseudo-random number generator by the independent streams, the initial bits in all the bit tables will have to be independent from each other, since the bits of subsequent generated elements are pushed onto the bit table. As such, when parallelizing an lagged Fibonacci generator using independent sequences, the stringent requirement for independence between the seed elements, $x_i, i = 1, 2, \ldots, p_1$, on each processor cannot be stressed any further.

To initialize the lag tables, most existing lagged Fibonacci generators, sequential and parallel, use an linear congruential generator to fill the elements of the lag tables. However, in `PLFG`, the lag tables were chosen to be initialized by a sequential pseudo-random number generator call Mersenne Twister [6]. The Mersenne Twister used has the Mersenne prime period, $\Pi_{MT19937} = 2^{19937} - 1$, thus known as the MT19937. This generator has passed several tests for randomness, including DIEHARD [6]. In addition, MT19937 has been tested to be very efficient, generating 10^7 pseudo-random numbers in 1.76 seconds on an Intel Pentium Pro 200MHz processor.

The lag values of `PLFG` were chosen to be $p_1 = 23209, p_2 = 9739$, recommended by Knuth in [3]. There is nothing in its design which prohibits the extension of the number of taps, n, to $n > 2$. The memory footprint is kept small, while maintaining high efficiency, by using a round-robin algorithm for lag table access.

Table 1. Results of 2D Ising model Monte Carlo simulation test with Metropolis algorithm; ϵ denotes error, σ denotes standard deviation, Cv denotes specific heat.

Generator	ϵ**energy**	σ**energy**	ϵ**Cv**	σ**Cv**
`PLFG`	0.0005960	0.0085104	0.0193600	0.0448184
SPRNG Multiplicative LFG	0.0091587	0.0269140	0.6682971	0.1442130
SPRNG Additive LFG	0.0188618	0.0193388	0.1587430	0.0911916
SPRNG Combined Multiple Recursive Generator	0.0678726	0.0216546	0.7692472	0.1732936

The output pseudo-random number is of type `unsigned long` in ANSI Standard C, which is typically a 32-bit data type on 32-bit architecture machines, but is a 64-bit data type on some 64-bit architecture machines. This dependence on machine architecture resulting in the variability in the period of the pseudo-random number sequence is indeed a feature since its period will automatically expand from when machine with wider word size becomes available. However, when used for simulations running on heterogeneous workstation clusters, this may be a concern.

The total number of independent streams is limited by the period of MT19937. Since total number of independent streams is $\frac{2^{19937}-1}{23209}$, this limitation is moot in practice. Thus, it is practically scalable to as many processors as needed. With $\Pi_{MT19937}$, Π_{MLFG} and p_1, all large, the probability that the sequences overlapping is minimal.

Both initialization and generation of the pseudo-random numbers can be performed in parallel, without any communication needed. The only time when communication is needed, is to synchronize before shutdown. This coarse-grained parallelism is highly desirable for Monte Carlo applications, which are themselves coarse-grained parallel as well.

5 Quality of Sequences Generated

`PLFG` has been subjected to the test using 2D Ising model Monte Carlo simulation with both the Metropolis and the Wolff algorithm.[1] The results shown in Tables 1 and 2 were obtained using identical test parameters. It is clear that the `PLFG` parallel pseudo-random number generator gives superior results compared to the results obtained by parallel pseudo-random number generators provided in the Scalable Pseudo-random Number Generator (SPRNG) package developed at the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.[2]

[1] The source code for the tests were ported from the Scalable Pseudo-random Number Generator (SPRNG) package.

[2] At the time of writing, SPRNG Version 2.0 has just been announced. The version of SPRNG considered here is Version 1.0.

Table 2. Results of 2D Ising model Monte Carlo simulation test with Wolff algorithm; ϵ denotes error, σ denotes standard deviation, Cv denotes specific heat.

Generator	ϵ**energy**	σ**energy**	$^{\epsilon}$**Cv**	$^{\sigma}$**Cv**
`PLFG`	0.0026430	0.0030601	0.0054660	0.0271471
SPRNG Multiplicative LFG	0.0541587	0.0287093	0.5353881	0.1165021
SPRNG Additive LFG	0.0660337	0.0291500	0.4078441	0.1705614
SPRNG Combined Multiple Recursive Generator	0.0130649	0.0105590	0.0444265	0.1105693

`PLFG` has also been put to test, against the multiplicative lagged Fibnacci generator from the SPRNG package, using the Relaxed Monte Carlo method for solving of systems of linear algebraic equations [10]. Comparing the results shown in Tables 3 and 4, it is clear that `PLFG` is at least on par with the multiplicative lagged Fibnacci generator from the SPRNG package, if not a better parallel pseudo-random number generator. The tests were done on a DEC Alpha XP1000 cluster, with EV67 processors running at 667MHz.

Table 3. Relaxed Monte Carlo method with `PLFG`, using 10 processors, on a DEC Alpha XP1000 cluster.

Data set	**Norm**	**Solution time (sec.)**	**RMS error**	**No. chains**
1000-A1	0.5	7.764	1.91422e-02	2274000
1000-A2	0.6	7.973	1.92253e-02	2274000
1000-A3	0.7	7.996	1.93224e-02	2274000
1000-A4	0.8	7.865	1.91973e-02	2274000
1000-A5	0.5	7.743	1.27150e-02	2274000
1000-A6	0.6	7.691	1.27490e-02	2274000
1000-A7	0.7	7.809	1.27353e-02	2274000
1000-A8	0.8	7.701	1.27458e-02	2274000

Timing tests for generating 10^6 pseudo-random numbers per stream have also been conducted. Results for tests conducted on both a cluster of DEC Alpha machines with Alpha 21164 500 MHz processors, connected via Myrinet, and a dual processor Intel x86 machine with Pentium Pro 200 MHz processors are shown in Table 5.[3] For tests on the DEC Alpha cluster, 20 processors were used. It can be seen that the speed of `PLFG` is on par with other parallel pseudo-random number generators.

[3] The Alpha 21164 and Pentium Pro 200 processors both have on-chip instruction and data L1 caches of 8Kb each, and 96Kb and 256 Kb L2 cache respectively.

Table 4. Relaxed Monte Carlo method with SPRNG MLFG, using 10 processors, on a DEC Alpha XP1000 cluster.

Data set	Norm	Solution time (sec.)	RMS error	No. chains
1000-A1	0.5	7.842	4.43195e-02	2274000
1000-A2	0.6	7.842	4.53718e-02	2274000
1000-A3	0.7	8.666	4.78022e-02	2274000
1000-A4	0.8	8.087	4.77088e-02	2274000
1000-A5	0.5	8.138	3.17604e-02	2274000
1000-A6	0.6	7.748	3.17574e-02	2274000
1000-A7	0.7	8.172	3.18349e-02	2274000
1000-A8	0.8	7.392	3.17931e-02	2274000

Table 5. Average time taken for generating 10^6 pseudo-random numbers.

Generator	Average time (sec.)	
	Intel Pentium Pro	DEC Alpha
`PLFG`	0.614	0.251
SPRNG Multiplicative LFG	0.720	0.187
SPRNG 64-bit LCG	1.510	0.078
SPRNG Additive LFG	0.270	0.260
SPRNG Combined Multiple Recursive Generator	2.910	0.238

6 Conclusion

`PLFG` is a highly efficient and scalable parallel pseudo-random number generator. Initialization of the lag tables can be sped up using another highly efficient pseudo-random number generator with good randomness qualities, while yielding quality sequences, as seen in the results of empirical tests conducted. In addition, coarse-grained parallelism employed in parallelizing `PLFG` and its scalability makes it extremely suitable for Monte Carlo simulations.

7 Acknowledgment

The author would like to thank J. A. Rod Blais from the Pacific Institute for the Mathematical Sciences, and Christiane Lemieux from the Department of Mathematics and Statistics, both at the University of Calgary, Canada, M. Isabel Casas Villalba from Norkom Technologies, Ireland, and Vassil Alexandrov from the High Performance Computing Center, University of Reading, UK, for their support and the fruitful discussions.

References

[1] Coddington, P. D. Analysis of Random Number Generators Using Monte Carlo Simulation. *International Journal of Modern Physics C5* (1994).

[2] Coddington, P. D. Random Number Generators for Parallel Computers. *National HPCC Software Exchange Review*, 1.1 (1997).

[3] Knuth, D. E. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*, 3 ed. Addison Wesley Longman Higher Education, 1998.

[4] L'Ecuyer, P. Maximally Equidistributed Combined Tausworthe Generators. *Mathematics of Computation 65*, 213 (1996), 203 – 213.

[5] Marsaglia, G. A Current View of Random Number Generators. In *Computing Science and Statistics: Proceedings of the XVI Symposium on the Interface* (1984).

[6] Matsumoto, M., and Nishimura, T. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation 8*, 1 (1998), 3 – 30.

[7] Niederreiter, H. *Random Number Generation and Quasi-Monte Carlo Methods*. No. 63 in CMBS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1992.

[8] Srinivasan, A., Ceperley, D., and Mascagni, M. Testing Parallel Random Number Generators. In *Proceedings of the Third International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing* (1998).

[9] Tan, C. J. K. Efficient Parallel Pseudo-random Number Generation. In *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications* (2000), H. R. Arabnia, et al., Ed., vol. 1, CSREA Press.

[10] Tan, C. J. K., and Alexandrov, V. N. Relaxed Monte Carlo Method for Solution of Systems of Linear Algebraic Equations. In *Recent Advances in Computational Science* (2001), V. N. Alexandrov, J. J. Dongarra, and C. J. K. Tan, Eds., vol. 2073 of *Lecture Notes in Computer Science*, Springer-Verlag.

[11] Vattulainen, I., Ala-Nissila, T., and Kankaala, K. Physical Models as Tests of Randomness. *Physics Review E52* (1995).

[12] Williams, K. P., and Williams, S. A. Implementation of an Efficient and Powerful Parallel Pseudo-random Number Generator. In *Proceedings of the Second European PVM Users' Group Meeting* (1995).