Implementation of an Efficient and Powerful Parallel Pseudo-random Number Generator

Kenneth P. Williams and Shirley A. Williams
University of Reading, Department of Computer Science
PO Box 225, Whiteknights, Reading, UK, RG6 6AY
{K.P.Williams,Shirley.Williams}@reading.ac.uk

Abstract

A reliable method of generating a high-volume of pseudo-random numbers is an essential requirement for any sort of stochastic modelling or Monte Carlo simulation work. We describe in detail the parallel PVM implementation of a pseudo-random number generator that combines the LeapFrog and Shuffling algorithms to create a single generator that preserves and extends the well-known randomness properties of these generators. The property of mutual disjointness between parallel sequences of random numbers (as guaranteed by the LeapFrog algorithm) is shown to only hold for one cycle of the base-generator being used. This limitation can be overcome by adding a shuffling routine which breaks up sequential correlations within each sequence. Hence the combined Shuffling-LeapFrog approach can be safely used beyond the first cycle of the base-generator while preserving the guarantee of producing mutually disjoint sequence of numbers. A PVM Master-Slave implementation of the method is described.

1 Introduction

Most scientific and engineering applications which use random numbers require that the generator being used to :

1. Produce a uniform distribution of pseudo-random numbers (i.e. uniform within a specific range).
2. Be reproducible (i.e. given the same seed, the same sequence of numbers will be reproduced.

3. Be easy to implement, fast and be memory efficient.

Uniform deviates are simply random numbers which lie within a pre-specified range (e.g. [0,1)), where all numbers within the range have an equal chance of being selected. The term is used to distinguish this form of distribution from other forms such as normal (Gaussian), poisson, or binomial distribution, etc. So a reliable source of random uniform deviates is an essential building block for any sort of stochastic modelling or Monte Carlo simulation work [7]. By far the most popular random number generators in use today are variations of the linear congruential method [4] which work as follows: first, choose values for the four parameters

m,	the modulus;	$m > 0$.
a,	the multiplier;	$0 \leq a < m$.
c,	the increment;	$0 \leq c < m$.
X_0	the starting value;	$0 \leq X_0 < m$.

The desired sequence of random numbers is then generated by the recurrence relation

$$X_{n+1} = aX_n + c \bmod m, n \geq 0. \tag{1}$$

The selection of these four parameters is clearly crucial to the performance of the generator and is discussed in detail elsewhere [3]. It can be shown that any sequence of numbers generated by a formula of the form $X_{n+1} = f(X_n)$ will always "get into a loop" (i.e. there is ultimately a cycle of numbers that is repeated endlessly). This repeating cycle is called a 'period'. Any useful sequence will have a relatively long cycling period.

2 Parallel Pseudo-random Number Generators

For the condition of 'mutual disjointness' across sequences of pseudo-random numbers generated in parallel to be true, no number in any sequence S^1 may occur in any other sequence S^i being generated in parallel, that is :

$$\bigcap_{i=1}^{n} S^i \equiv S^1 \cap S^2 \cap \ldots \cap S^n = \emptyset \tag{2}$$

where n is the number of sequence of pseudo-random numbers being generated in parallel. This important theoretical property [5] is useful in many applications.One of the attractions of the LeapFrog method is its' ability to guarantee the parallel generation of mutually disjoint sequences of pseudo-random numbers.

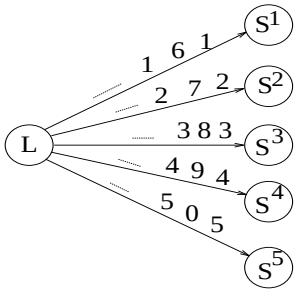

Figure 1: The LeapFrog method for generation of pseudo-random numbers.

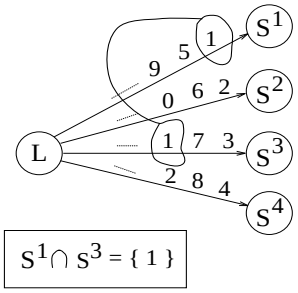

Figure 2: The LeapFrog method loses mutual disjointness after first cycle.

2.1 The Shuffling-LeapFrog Method

The LeapFrog method of generating parallel sequence of pseudo-random numbers [2] is a simple yet effective approach. A base-generator[1] is used to generate a sequence of pseudo-random numbers. As each number is produced it is sent to one of the 'slave' processes which are serviced in turn by the generator process, L, (see Fig.1). Hence slave S^1 receives the first number in the sequence, S^2 receives the second, and so on, until each slave has been sent a number. The generator then returns to the first slave and sends it the next number in its' sequence.

The problem with the LeapFrog algorithm however, is that once one cycle of the base-generator has been completed the guarantee of mutual disjointness of sequence of random numbers no longer holds, (see Fig. 2), unless condition (3) is satisfied :

$$p \bmod N = 0 \tag{3}$$

[1] often a system-defined routine such as *drand*48() in Unix.

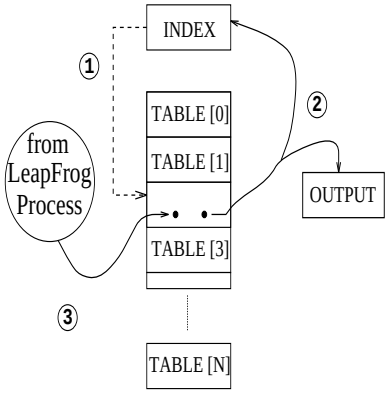

Figure 3: The Table-Shuffling method for breaking up sequences of numbers.

where : p is the period length of the base-generator, and N is the number of slave processes. The shuffling method is used to break up sequential correlations in sequence of incoming numbers. In Fig. 3, circled numbers indicate the sequence of events : on each call, the value of *index* is used to select a random element from the table. That element becomes the output random number and is also the next value of the *index* variable. Its place in the table is refilled by the next number from the incoming sequence.

The two methods can be combined by introducing a shuffling routine for each sequence of numbers generated by the LeapFrog process. This has the effect of maintaining mutual-disjointness across sequences (as guaranteed by the LeapFrog method) while breaking up correlation of sequences of numbers within each sequence.

3 PVM Master / Slave Implementation

A simple Master/Slave model was chosen to implement the new method with PVM. In our original implementation the Master process spawned 'N' slave processes and one 'L' (LeapFrog generator process). A single seed was then sent to the L process which started generating sequences of pseudo-random numbers and distributing them among the slaves. Each slave performs its own shuffling routine on its incoming sequence of numbers before using the numbers for whatever purpose they are required. Rather than send one number at a time from L to each slave we decided to send numbers in 'blocks' of 100, this significantly reduced the communication overhead. This still left us with the problem that if the L process was placed on a particularly slow processor then the whole program would run slowly (i.e. a 'bottleneck' would occur).

These problems were overcome in our second implementation. Here, the

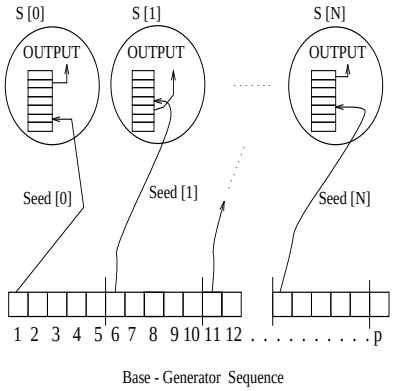

Figure 4: Each slave starts from a unique point in the base-generator sequence.

Master spawned N slaves as before but now also calculates a unique *seed* for each slave. No L process is required since now each slave has its own LeapFrog routine. The Master sends two numbers to each slave (i) its unique *seed*, and (ii) a block-size value b. The seed values are taken at equi-distant points along the pseudo-random sequence generated by the base-generator (see Fig 4). Each $seed^i$ for each $slave^i$ can be directly calculated from the base-generator sequence using a generalisation of (1)

$$X_{n+i} = (a^i X_n + (a^i - 1)c/(a-1)) \bmod m, \text{ where } n \geq 0, i \geq 0. \qquad (4)$$

and provided that $0 \geq (n+k) < m$. The block-size value b is calculated by

$$b = \lfloor p/N \rfloor \qquad (5)$$

Starting from $seed^i$ each $slave^i$ will then generate the next b elements in the sequence feeding them into its own shuffling routine from where pseudo-random numbers may be extracted by the application, as required. Having generated b elements, the slave process can then "re-seed" itself with the same seed value and repeat it's process of generation, for as long as necessary.

4 Conclusions

In conclusion we note that the Shuffling-LeapFrog method can easily be made portable by not using any system-defined random number generators for the base-generator required by the LeapFrog algorithm. Preliminary application of spectral [3], chi-squared (χ^2), and random-walk tests indicate that the Shuffling-LeapFrog method scores highly on the criteria examined by these tests. As such, detailed comparisons with other parallel pseudo-random number generators will be possible.

The addition of the shuffling routine to the basic LeapFrog method significantly extends the period of the generator. Once the base-generator has cycled the numbers output by each slave will overlap with numbers which that particular slave has already produced, the sequencing however, will be different. In order for a slave to start cycling in the traditional sense, its' entire "state" has to be replicated (i.e. the entire contents of its' shuffle-table, the same values in the same locations, and also the same *index* value). Furthermore, this state also has to be reproduced at precisely the same point in the input sequence as when it occurred previously, making the Shuffling-LeapFrog technique a powerful method for generating a high volume of pseudo-random numbers in parallel.

References

[1] Bays,C.,& Durham,S.D., *in* ACM Trans. Mathematical Software, 2, 59-64, (1976).

[2] Foster,I., "Designing and Building Parallel Programs", Addison-Wesley (1995).

[3] Knuth,D.E., "The Art of Computer Programming" - Vol. 2 (Semi-Numerical Algorithms), Chapter 3, (2nd Edition), Addison-Wesley, (1981).

[4] Lehmer,D.H., see "Proc. of 2nd Symp. on Large-Scale Digital Calculating Machinery", Cambridge:Harvard University Press, 141-146, (1951).

[5] McLaren,N., "The Generation of Multiple Independent Sequences of Pseudo-random Numbers", Applied Statistics, 38, 351-359, (1989).

[6] Press,W.H, Flannery, B.,Teukolsky,S.A.,& Vetterling,W.T., "Numerical Recipes in C", Cambridge University Press, (1990).

[7] Williams,K.P.,Williams,S.A.,& Mitchell,P., "Monte Carlo Simulations of Polynuclear Growth Mechanisms", to appear, $PARCO-95$, Universiteit Gent, Belgium, Sept. 19-22, (1995).