MODERN GENERATORS OF MULTIPLE STREAMS OF PSEUDO-RANDOM NUMBERS

Krzysztof Pawlikowski and Marcus Schoo
Department of Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand
E-mail: krys.pawlikowski@canterbury.ac.nz

Donald C. McNickle
Department of Management
University of Canterbury
Christchurch, New Zealand

ABSTRACT

In this paper, we examine the required features of modern PPRNGs (Parallel Pseudo-Random Number Generators), from the point of view of their possible applications in the Multiple Replications in Parallel (MRIP) paradigm of stochastic simulation. Having surveyed the most recommended generators of this class, we test their implementations in C/C++. Their performance is compared from the point of view of initialization and generation times and results are given detailing the fastest and slowest.

1 INTRODUCTION

Parallel PRNGs (PPRNGs) with the required empirical, analytical and deterministic properties are not trivial to find (Hellekalek 1998, Mascagni & Srinivasan 2000). However, recent research has resulted in several generators which appear to be of high quality (Fischer, Carmon, Ariely, Zauberman & L'Ecuyer 1999, Mascagni & Srinivasan 2000, Mascagni & Srinivasan 2004, Matsumoto & Nishimura 1998*a*, Panneton & L'Ecuyer 2005).

In this paper, we will examine the required features of modern PPRNGs, from the point of view of their possible applications in Multiple Replications in Parallel paradigm of stochastic simulation(G. Ewing 1997), in which each simulation engine runs an independent version of the simulation and submits results of observations regularly to a central processor for analysis. The MRIP paradigm of simulation has become more popular with emergence of such packages as AKAROA-2, a fully automated simulation tool for running stochastic simulation in the MRIP paradigm, developed at the University of Canterbury, Christchurch, New Zealand(G.C.Ewing & D.McNickle 1999).

As MRIP distributes identical replications of a given simulation over different simulation engines, each engine requires an independent sequence of Pseudo-Random Numbers (PRNs)(G.C.Ewing & D.McNickle 1999). The success of such a simulation is highly dependent on the quality of these multiple independent streams and the efficiency of their generation. The PPRNGs require the following properties: Intra-Stream Uniformity and Independence; Inter-Stream Independence; Satisfactorily many satisfactorily long streams of PRNs; Efficient Implementation.

Given a single stream of PRNs we can apply tests, such as those described, for example, by Knuth (1997), to ascertain if we are confident that the first property is satisfied.

Being satisfied that we have a generator that is able to produce a single stream of i.i.d random variables, two paradigms exist for generating parallel streams.

Cycle splitting is the method of taking a single stream $\{x\}$ of PRNs produced by a single generator and splitting this stream into P sub-streams $\{\{x^1\}, \{x^2\}, \ldots, \{x^P\}\}$, where P is the number of processors used in a simulation. There are two variations on this paradigm. In *blocking* we determine a block size B and assign to the ith processor the stream $\{x^i\} = \{x_{iB}, x_{iB+1}, \ldots, x_{iB+(B-1)}\}$. Alternatively, in *leap-frog*, if P is the number of processors, we produce the stream for the ith processor as $\{x^i\} = \{x_i, x_{i+P}, x_{i+2P}, \ldots\}$. Other methods exist but all of them involve distributing the finite sequence produced by one generator to P processors. A potential problem is that PRNs generated by linear generators can experience long range correlations (Mascagni & Srinivasan 2004). Under cycle splitting such long range correlations can introduce short range inter-stream correlations when using *blocking* or short range intra-stream correlations when using *leap-frog*(Mascagni & Srinivasan 2004).

An alternative to *cycle-splitting* is *parameterization*, the method of creating a new, full cycle, independent generator for each processor from a family of generators. *Seed Parameterization* is used with generators that produce independent full length cycles depending on the seed value. *Iteration Function parameterization* modifies some value within the iteration function so that each processor uses a different generator. The limiting factor here is how many independent streams the parameterization method can produce for a particular generator and how

quickly it can produce them.

According to a recently established theoretical restriction for two dimensional pseudo-uniformity, a PRNG generating numbers in a cycle of length L should be used in a single simulation as a source of not more than $16\sqrt[3]{L}$ numbers in the case of linear generators. Assuming that only 1% of simulation time is spent on generating PRNs, a computer with a CPU clock running at 100THz (achievable by 2042 assuming Moore's Law with clock speed doubling each 1.5 years) would need a linear PRNG with cycle length of about 2^{110}, for executing a simulation lasting one hour, or about 2^{140} for executing a week long simulation. If a PRNGs cycle is split into say $2^{14} = 16384$ substreams for a parallel simulation, then such a PRNG should have a cycle length of about 2^{160} for executing a simulation lasting one hour, or about 2^{182} for executing a week long simulation.

A modern PPRNG must satisfy all the properties described above and several have been proposed that do so. However, as well as being of long period and statistically robust, there must be an efficient implementation both in terms of memory and speed. In section 2 we introduce the generators that we consider. Section 3 describes our experiments and the results.

2 PARALLELIZABLE GENERATORS

As distributed (and parallel) computing has become more available and popular the need for PPRNGs that satisfy the requirements described above has increased. Many PRNGs with parallelization techniques have been proposed. Linear Congruential Generators (LCGs) and Feedback Shift Register Generators have received the most attention and the theory and implementations of these generators is highly developed. The history and properties of two such generators of particular interest to distributed applications are described in sections 2.1 and 2.2. Many alternatives exist however.

Mascagni and Srinivasan (2000) describe methods to parameterize the simple linear congruential generator $x_n = ax_{n-1} + b \pmod m$ by way of varying a when m is prime, or b when m is a power of 2. The period of such methods is limited by the modulus to $m - 1$ and m respectively. Disadvantages, including poor randomness in the least significant bits, make LCGs with $m = 2^k$ a poor choice(Mascagni & Srinivasan 2000). The alternative, LCGs with prime moduli, are most often implemented with a Mersenne prime modulus as a fast algorithm exists for modular multiplication with Mersenne prime moduli. However, to find suitable values for a we must know all primitive roots of m, a computationally complex task. To make the calculation of primitive roots trivial, Mascagni and Chi proposed an LCG family with Sophie-Germain prime (Sophie-Germain primes are of the form $m = 2p + 1$ where m and p are prime) moduli and a fast modular multiplication algorithm for Sophie-Germain primes(Mascagni & Chi 2004). They implemented this method in the form of their Sophie-Germain Modulus Linear Congruential Generator(SGMLCG), a 64-bit LCG family capable of producing up to $2^{63} - 10509$ independent full period generators with period $2^{63} - 21016$. SGMLCG has passed Marsaglia's Diehard tests (Marsaglia 1995) as well as the tests given as part of Mascagni's SPRNG package (Mascagni & Srinivasan 2000).

An alternative to linear PRNGs are the Inversive Congruential Generators (ICGs) and Explicit Inversive Congruential Generators (EICGs) by Eichenauer et al.(1986, 1993). They have similar properties to linear congruential generators but have the distinct advantage of the absence of the lattice structure associated with linear generators. They do, however, have the disadvantages or requiring modular inversion, a computationally costly process. For further discussion on the theoretical and empirical properties of inverse generators as well as splitting techniques we refer to (Hellekalek 1995).

2.1 MRG32k3a - Combined Multiple Recursive Generator

Linear Congruential Generators, first put forward by Lehmer (1951) in 1949, are based on the following simple linear recurrence;

$$x_n = ax_{n-1} + b \pmod m \quad (1)$$
$$u_n = \frac{x_n}{m} \quad (2)$$

so that u_n is a i.i.d random variable in the range $[0, 1)$.

Early questions on the statistical robustness of LCGs, as well their relatively small maximum period of $p = m$ ($p = m - 1$ for $b = 0$), prompted research into the *multiple recursive generator (MRG)*(Grube 1973), defined as follows;

$$x_n = a_1x_{n-1} + a_2x_{n-2} + \ldots + a_kx_{n-k} \pmod m \quad (3)$$
$$u_n = \frac{x_n}{m} \quad (4)$$

When the characteristic polynomial $P(z) = z^k + a_1z^{k-1} + a_2z^{k-2} + \ldots + a_k$ is primitive, the MRG achieves its maximum period of $p = m^k - 1$, a considerable improvement on a LCG with equal modulus. To achieve a high quality in the sense of the *spectral test*, the coefficients of recurrence (3) must be chosen such that $\sum_{i=1}^{k} a_i^2$ is large. However for sake of efficiency we wish to keep these coefficients small. To manage this conflict, L'Ecuyer (1996) introduced the combined MRG (CMRG) which combines J MRGs as follows;

$$x_{j,n} = a_{j,1}x_{j,n-1} + \ldots + a_{j,k_j}x_{j,n-k_j} \pmod{m_j} \quad (5)$$
$$u_n = \left(\sum_{j=1}^{J} \frac{x_{j,n}}{m_j}\right) \pmod 1 \quad (j = 1, 2, \ldots, J) \quad (6)$$

L'Ecuyer et al.(1996, 1999) investigated many parameters to make up the components of this CMRG and has published examples of the CMRGs that are statistically robust, easy to implement, efficient to run and generate

PRNs in very long cycles. One such example is the so called MRG32k3a which is defined by two MRGs with three terms each. This MRG32k3a has been shown by L'Ecuyer et al.(1999) to perform well in statistical tests up to at least 45 dimensions. It has a period of $p \approx 2^{191}$, achievable with arbitrary seed initialization with at least one non-zero element in each MRG.

An attractive feature of LCGs is that we can easily *fast forward* the generator k steps as follows:

$$x_{n+i} = a^i x_n \pmod{m} \quad (7)$$

This property allows us to easily perform a similar operation on the MRGs which make up MRG32k3a and parallelize via the *blocking* paradigm(L'Ecuyer, Simard, Chen & Kelton 2001). If we put $s_{j,n}$, the nth state of the jth MRG, as the vector $\{x_{j,n}, x_{j,n+1}, x_{j,n+3}\}$ then a 3×3 matrix A exists such that,

$$s_{j,n+1} = A_j s_{j,n} \pmod{m_j} \quad (8)$$

And so the state $s_{j,n+i}$ is given by,

$$s_{j,n+i} = A_j^i s_{j,n} \pmod{m_j} \quad (9)$$

This approach is taken in the implementation given in (L'Ecuyer et al. 2001). The implementation takes the full cycle and splits it into streams of length 2^{127}, splitting each of those into 2^{51} sub-streams of length 2^{76}.

2.2 Dynamic Creation of Mersenne Twisters

As an alternative to early poor LCGs, the theory of Feedback Shift Registers (FSRs) was developed. Such generators offered better randomness than LCGs and, as they worked using only bitwise operations, were very fast. 1973 saw the introduction of the *Generalized FSR (GFSR)* PRNGs (Lewis & Payne 1973) which were based on the following recurrence;

$$\vec{x}_{l+n} = \vec{x}_{l+m} \oplus \vec{x}_l, \quad (l = 0, 1, \ldots) \quad (10)$$

where $\vec{x}_i$ is the ith word vector of size w and $\oplus$ is binary addition (exclusive OR). GFSRs were generalized in the sense that, with suitably chosen seed, the period $2^n - 1$ was not reliant on word size of the machine but on n, the number of words used to store the state of the generator, which allowed for arbitrary long periods. Matsumoto and Kurita (1992) recognized the merits of GFSR but also identified four disadvantages: selection of seed is difficult; randomness qualities are questionable as it is based on the trinomial $t^n + t^m + 1$; GFSRs period of $2^n - 1$ is much smaller than the theoretical maximum of 2^{nw}; and n words of memory are required to produce a period of $2^n - 1$. To address these disadvantages Matsumoto and Kurita (1992) developed the *Twisted GFSR (TGFSR)* PRNG which introduced a twisting matrix $A(w \times w)$ into the GFSR recurrence as follows:

$$\vec{x}_{l+n} = \vec{x}_{l+m} \oplus \vec{x}_l A, \quad (l = 0, 1, \ldots) \quad (11)$$

For appropriately chosen values of n, m and A, TGFSR achieves the maximal period of $2^{nw} - 1$ and, due to the properties of the twisting matrix, achieves better randomness, as the recurrence represents a primitive polynomial with many terms rather than a trinomial. With these advantages such generators were able to be created with periods never before seen, such as the popular T800 with period 2^{800}. Despite these successes, the inclusion of the A matrix in TGFSR introduced a defect in k-distribution for k larger than the order of the recurrence(Matsumoto & Kurita 1994). The difficulty stemmed from trying to set A such that $\vec{x}_l A$ was fast to calculate and to have good k-distribution for large k. To address this difficulty a *tempering* matrix T was introduced as follows;

$$\vec{x}_{l+n} = \vec{x}_{l+m} \oplus \vec{x}_l A, \quad (l = 0, 1, \ldots) \quad (12)$$
$$\vec{z}_{l+n} = \vec{x}_{l+n} T \quad (13)$$

which, for appropriate values for T, is equivalent to using a more computationally complex A. In (12) x_{l+n} is the output which is used in further recursions whereas z_{l+n} is the output random variable.

Testing the characteristic polynomial of A for primitivity requires the complete factorization of $2^{mw} - 1$(Matsumoto & Kurita 1992). For many large nw such decompositions are not known, and thus a limited number of large period TGFSRs were possible. To address this limitation Matsumoto and Nishimura(Matsumoto & Nishimura 1998*b*) invented the Mersenne Twister by adjusting the recurrence (12) to allow for a Mersenne prime period as follows:

$$\vec{x}_{k+n} = \vec{x}_{k+m} \oplus \left(\vec{x}_k^u \mid \vec{x}_{k+1}^l\right) A, \quad (k = 0, 1, \ldots) \quad (14)$$
$$\vec{z}_{k+n} = \vec{x}_{k+n} T \quad (15)$$

such that $nw - r$ is the size of the state array of the generator and 2^{nw-r} is a Mersenne prime. For a predetermined integer $r (0 \leq r \leq w - 1)$ the designation $\left(\vec{x}_k^u \mid \vec{x}_{k+1}^l\right)$ means the concatenation of $\vec{x}_k^u$ (the upper $w - r$ bits of $\vec{x}_k$) and $\vec{x}_{k+1}^l$ (the lower r bits of $\vec{x}_{k+1}$).

In the same paper as the Mersenne Twister algorithm, code was released for MT19937, a Mersenne Twister PRNG with a period of $2^{19937} - 1$ and 623-dimensional equidistribution up to 32-bit accuracy. Such a massive period is possible as the prime decomposition of a Mersenne prime is trivial and so the testing of primitivity of polynomials becomes much faster(Matsumoto & Nishimura 1998*b*). MT19937 has passed empirical tests such as Marsaglia's Diehard tests(Marsaglia 1995) and Load and Ultimate Load Tests executed by the pLab group (pLab 2004).

No algorithm is currently known for *fast forwarding* MT19937 in a similar way that exists for MRG32k3a, so cycle splitting is not a good option. However, a Mersenne Twister is able of being parallelized through a parameterization technique called Dynamic Creation as described and implemented by Matsumoto and Nishimura (1998*a*). The published implementation allows creation of up to

$2^{\frac{w}{2}}$ parallel Mersenne twisters with periods including Mersenne primes from $2^{521}-1$ to $2^{44497}-1$.

Despite their advantages, Mersenne Twister PRNGs have one notable flaw. The recurrence (14) modifies very few bits at each step and as such a poor distribution in the state array will have long lasting affects in that the poor distribution will remain in the state array for many subsequent states(Panneton & L'Ecuyer 2005). As such, we must take care when initializing the seed of Mersenne Twister as a poor seed may produce a long non-random stream of numbers. This is solved in the implementation of MT19937 and Dynamic Creation by having a LCG to randomly initialize the seed. This is, however, a poor solution to the problem for two reasons. One, we prefer generators that produce i.i.d numbers for any arbitrary choice of seed (other than all 0's). Two, the massive period of Mersenne Twister is achieved as every permutation of the state array occurs somewhere in the period. As such, even if the seed is not a poor state, the poor states will occur somewhere in the period. Due to the super-astronomical period of the Mersenne Twister we are not likely to reach this poor state during any conceivable application so this is a theoretical consideration only. To deal with this problem Panneton and L'Ecuyer have developed WELL generators, an improved long-period generator class based on linear recurrences modulo 2. While these WELL generators perform much better in terms of recovering from a poor state(Panneton & L'Ecuyer 2005), they will still produce a stream of poorly distributed numbers given a bad state (though much shorted than Mersenne Twister). As such initialization of seed should still be done with care and further development of this class of generators to address this problem is needed before arbitrary seed choice is a reality.

2.3 Multiplicative Lagged-Fibonacci Generators

In an effort to achieve larger periods than offered by LCGs, researchers in the 1950s considered recursions which based x_n not only on x_{n-1} but also on x_{n-2} as follows(Knuth 1997):

$$x_n = ax_{n-1} + cx_{n-2} \pmod{2^b} \tag{16}$$

The period of such a generator, for appropriate values of a, c and $m = 2^b$ is as high as $m^2 = (2^b)(2^b)$, a marked improvement over LCGs with maximum period $m = 2^b$. In the simplest case when $a = c = 1$ the recurrence (16) represents the Fibonacci recurrence which displays poor randomness qualities. To improve on this, a lag l was added to the Fibonacci recurrence as follows(B.F. Green, Smith & Klem 1959):

$$x_n = x_{n-1} + x_{n-l} \pmod{2^b} \tag{17}$$

For large values of l ($l > 15$), (17) achieves much better randomness than (16). Another advantage is that the period, $p = (2^l - 1)(2^{b-1})$, is dependent on l as well as b. As such, the period of such a generator can easily be increased by choosing a larger lag l.

Even better performance is achieved supplementing the *lag* l in (17) with a *short lag*, k, as follows,

$$x_n = x_{n-k} + x_{n-l} \pmod{2^b} \tag{18}$$

such that $k < l$. For appropriate values the maximum period of $p = (2^l - 1)(2^{b-1})$ is achieved. (18) forms the so called *additive lagged-Fibonacci generator (ALFG)*. The ALFG has been used extensively though it is now recognized that it performs poorly in some relative simple statistical tests for even relative large lags(Knuth 1997). As such it is necessary to choose l to be very large to ensure a robust generator(Mascagni & Srinivasan 2004). An alternative that performs much better is the *multiplicative lagged-Fibonacci generator (MLFG)* defined by the following recurrence,

$$x_n = x_{n-k} \times x_{n-l} \pmod{2^b} \tag{19}$$

The MLFG demonstrates better robustness than the ALFG (Mascagni & Srinivasan 2004), however several features of this generator are noteworthy. Firstly, it has a slightly smaller period than the ALFG, with a maximum period of $p = (2^l - 1)(2^{b-3})$, for appropriate values of k and l and seed $(x_{n-1} \ldots x_{n-l})$. Secondly, due to the multiplicative nature of the generator, and the fact that an odd number multiplied with an even number gives an even number, the sequence produced by (19) will eventually become all even, if not all seed values are odd. To avoid this transient period at the beginning of the stream, it is recommended to seed a MLFG with all odd values. Further to this, the user must recognize that the least significant bit of all numbers produced by MLFGs seeded in this way will always be 1(Mascagni & Srinivasan 2004).

In terms of parallelization, cycle splitting and parameterization algorithms exist for both the ALFGs and MLFGs. Efficient cycle splitting via blocking for both ALFGs and MLFGs have been presented by Makino (1994). Mascagni et al.(1995) proposed a parallelization of ALFGs based on seed parameterization capable of yielding $2^{(b-1)(l-1)}$ distinct maximum period cycles. A similar technique for MLFGs was proposed by Mascagni and Srinivasan (2004) yielding $2^{(b-3)(l-1)}$ uncorrelated cycles. The default lag in Mascagni implementation is $l = 17$ with $b = 64$, giving 2^{976} streams, each of period approximately 3×2^{76}. Similar to their sequential versions, the parameterized MLFGs display better robustness than the parameterized ALFGs with the latter failing some standard tests due to inter-stream and intra-stream correlations. The MLFGs perform well in tests, however, even with small lags(Mascagni & Srinivasan 2004).

3 EMPIRICAL COMPARISON

We recognize that cycle splitting with MRG32k3a, Dynamic Creation of Mersenne Twisters (if appropriately

initialized) and seed parameterization with MLFGs produce high quality parallel streams in terms of inter-stream and intra-stream independence, period length and number of possible streams. However, we wish to consider the efficiency of these generators' published implementations with respect to their applications in the MRIP paradigm of stochastic simulation.

3.1 Platform

Experiments were conducted on a Intel Pentium 4 CPU running at 2.4GHz with 512KB of cache, a floating point unit and 512MB of RAM, running Linux version 2.4.20-24.9, gcc version 3.2.2 and Red Hat 9. As the generators were tested for their use in practical applications, it is unrealistic to expect that the majority of users would re-implement these generators themselves. As such published implementations of the generators were used. A C++ object oriented implementation of MRG32k3a was made available by L'Ecuyer at (L'Ecuyer 2006). A C implementation of Dynamic Creation was made available by Matsumoto at (Matsumoto 2004). SPRNG(Mascagni 2005) (Scalable Pseudo-Random Number Generators) is a library of tested parameterizable generators by Mascagni, which includes the MLFG used here. For the purpose of comparison we included in our tests a combined EICG with a period of $P = 2^{93}$, implemented through the pLab's PRNG library(pLab 2004).

3.2 Initialization

Before taking part in a simulation each engine must be assigned an independent stream of PRNs. We consider initialization to be this process of assigning a stream to an engine, including any calculation of parameters, initialization of seeds etc. Initialization is the time taken from the instant when an engine requests a stream to the point at which it is able to start generating PRNs. Figure (1) shows a comparison between the four generators we consider. For each generator we initialize n streams, for various n, and plot the average initialization time for that n. For example if $n = 3$ and a given PPRNG finishes initializing the first stream after 2 seconds, the second after 4 seconds and the third after 6 seconds, the mean initialization time for that generator at $n = 3$ would equal 4. As each engine may begin simulation as soon as its stream is initialized, the mean time is most appropriate for comparison. Initialization of cEICG and MRG32k3a is done using a fast forwarding technique which is sequential. This means as n becomes large the average time needed to initialize a stream increases linearly. While both processes are linear, we see the initialization by cEICG is slower than MRG32k3a. The implementation of MRG32k3a assumes fixed stream sizes and as such has precomputed the matrix required to move from one stream to the next. The pLab's cEICG allows for arbitrary stream sizes so no precomputation is possible and so initialization is much slower. Both Dynamic Creation and

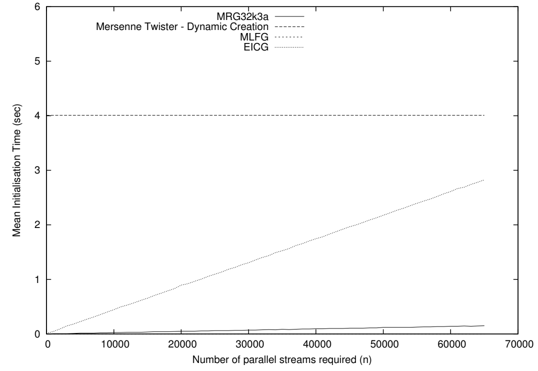

Figure 1: Mean waiting time for stream initialization

the SPRNG MLFG follow the parameterization paradigm and both implementations have initialization routines that accept an ID and return independent streams for independent IDs. Working on the assumption that each engine has access to a unique ID, this allows the initialization of Dynamic Creation and SPRNG MLFG to be parallelized in such a way that each engine may initialize it's own generator concurrently without any inter-engine or engine-control unit communication and without fear of loss of independence. As such, the average initialization time of these generators is constant as n increases. To reach a figure for these constants 100 Mersenne Twisters and 100,000 MFLG streams were created and the mean time was taken. Due to the fast constant initialization time of the MLFG its line is not visible, running along the x-axis at a time of 0.0001 seconds.

MRG32k3a streams were initialized to be the default length of 2^{127}. Dynamic Creation was asked to create Mersenne Twisters with 32 bit word length and period 2^{521}, an implementation minimum. SPRNG created MLFGs with the default lag of $l = 17$ and $b = 64$, yielding generators of period $p \approx 3 \times 2^{76}$. PLab's PRNG library was asked to construct CEICG streams of length $p \approx 2^{62}$.

3.3 Generation

We assume that, once initialization is complete, each engine will have access to an independent stream of PRNs at least 2^{76} numbers in length. As such, it is practically impossible for an engine to exhaust its allocated stream and require another. Having looked at initialization speed we need only to look at the speed at which PRNGs generate numbers.

All four generators were required to generate n numbers in the range $[0, 1)$. Figure (2) shows generation times for all generators tested. As expected all generators run in linear time with respect to n. As all operations in Dynamic Creation's Mersenne Twisters are bitwise, it is the fastest. A single multiplication and modulo operations mod 2 make MLFG the second fastest. The more complex operations of MRG32k3a make it the third fastest,

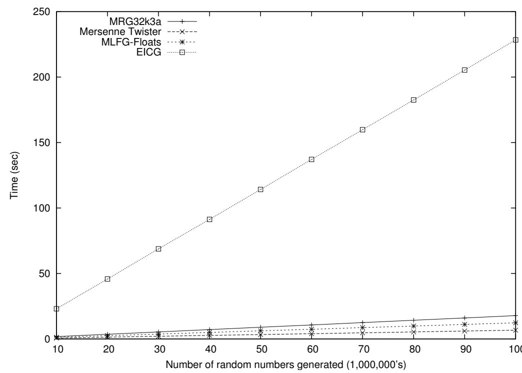

Figure 2: Generation Time per Simulation Engine

while the inversion operation of the cEICG make it by far the slowest. The performance of the cEICG can be improved by reducing the number of EICGs that make up the cEICG. However, this will also reduce to period of the generator.

4 CONCLUSION

Given the existence of a single stream PRNG, two paradigms exist which may be employed to achieve parallel streams of PRNs. Cycle splitting, which involves distributing a single large cycle into various streams. Alternatively, parameterization involves modifying some parameter in the base generator such that different parameters result in different independent full period streams.

Though high quality generators are difficult to find, several have been proposed. We investigated several that are potentially useful in massively parallel stochastic simulations under the MRIP scenario. L'Ecuyer's MRG32k3a PRNG is a large period linear generator. It achieves its large period and good randomness by combining several Multiple Recursive Generators and is parallelized by the cycle splitting paradigm. Matsumoto and Nishimura's Dynamic Creation generates independent Mersenne Twister PRNGs by parameterizing the matrix A in recurrence (14). Mascagni and Srinivasan's parameterization of the Multiplicative Lagged-Fibonacci Generator is implemented within the SPRNG library and is based on seed parameterization. We also consider the Explicit Inversive Congruential Generators as implemented within the pLab's PRNG library. However, despite excellent randomness properties, the last class of PRNGs is significantly slower than the other three generators considered.

The generators were tested for initialization and generation speed to assess the efficiency of current implementations. Initialization was completed most quickly by the Multiplicative lagged-Fibonacci Generator and most slowly by Dynamic Creation. Generation of numbers was performed most quickly by Dynamic Creation's Mersenne Twisters and most slowly by the Explicit Inversive Congruential Generator.

5 ACKNOWLEDGMENTS

This work was supported by the University of Canterbury, New Zealand, Summer Scholarship (U1042). The author(s) wish to thank Makoto Matsumoto and Pierre L'Ecuyer for their correspondence assisting the writing of this report. The first author would like to thank his parents for their encouragement and support and Jennifer for her inspiration and understanding.

References

B.F. Green, J., Smith, J. E. K. & Klem, L. 1959, 'Empirical tests of an additive random number generator', *J. ACM* **6**(4), 527–537.

Eichenauer-Herrmann, J. 1993, 'Statistical independence of a new class of inversive congruential pseudorandom numbers', *Math. Comp.* **60**, 375–384.

Eichenauer, J. & Lehn, J. 1986, 'A non-linear congruential pseudo random number generator', *Statist. Papers* **27**, 315–326.

Fischer, G., Carmon, Z., Ariely, D., Zauberman, G. & L'Ecuyer, P. 1999, 'Good parameters and implementations for combined multiple recursive random number generators', *Operations Research* **47**(1), 159–164.

G. Ewing, D. McNickle, K. P. 1997, Multiple replications in parallel: Distributed generation of data for speeding up quantitative stochastic simulation, *in* 'Proceedings of 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics', Berlin, pp. 379–402.

G.C.Ewing, K. & D.McNickle 1999, Akaroa-2: Exploiting network computing by distributing stochastic simulation, *in* 'Proceedings of European Simulation Multiconference ESM'99', Int. Society of Computer Simulation, Warsaw, Poland, pp. 175–181.

Grube, A. 1973, 'Mehrfach rekursiv-erzeugte pseudozufallszahlen', *Zeitschrift für angewandte Mathematik und Mechanik* **53**, T223–T225.

Hellekalek, P. 1995, Inversive pseudorandom number generators: concepts, results, and links, *in* C. Alexopoulos, K. Kang, W. Lilegdon & D. Goldsman, eds, 'Proceedings of the 1995 Winter Simulation Conference', pp. 255–262.

Hellekalek, P. 1998, Good random number generators are (not so) easy to find, *in* 'Selected papers from the 2nd IMACS symposium on Mathematical Modelling—2nd MATHMOD', Elsevier Science Publishers B. V., pp. 485–505.

Knuth, D. 1997, *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*, Addison-Wesley Longman Publishing Co., Inc.

L'Ecuyer, P. 1996, 'Combined multiple recursive generators', *Operations Research* **44**(5), 816–822.

L'Ecuyer, P. 2006, 'http://www.iro.umontreal.ca/ lecuyer/'.

L'Ecuyer, P., Simard, R., Chen, E. & Kelton, W. 2001, 'An object-oriented randomnumber package with many long streams and substreams'.

Lehmer, D. H. 1951, Mathematcial methods in large-scale computing units, *in* 'Proc. 2nd Symp. on Large-Scale Digital Calculating Machinery', Harvard University Press, Cambridge, Mass., pp. 141–146.

Lewis, T. G. & Payne, W. H. 1973, 'Generalized feedback shift register pseudorandom number algorithm', *J. ACM* **20**(3), 456–468.

M. Mascagni, S.A. Cuccaro, D. P. & Robinson, M. L. 1995, 'A fast, high quality, and reproducible parallel lagged-fibonacci pseudorandom number generator', *Computational Physics* **119**, 211–219.

Makino, J. 1994, 'Lagged-fibonacci random number generators on parallel computers', *Parallel Computing* **20**, 1357–1367.

Marsaglia, G. 1995, Diehard software package. ftp://stat.fsu.edu/pub/diehard.

Mascagni, M. 2005, 'The scalable parallel random number generators library (sprng) for asci monte carlo computations'. http://sprng.cs.fsu.edu/.

Mascagni, M. & Chi, H. 2004, 'Parallel linear congruential generators with sophie-germain moduli', *Parallel Computing* **30**, 1217–1231.

Mascagni, M. & Srinivasan, A. 2000, 'Sprng: A scalable library for pseudorandom number generation', *ACM Transactions on Mathematical Software* **26**(3), 436–461.

Mascagni, M. & Srinivasan, A. 2004, 'Parameterizing parallel multiplicative lagged-fibonacci generators', *Parallel Computing* **30**, 899–916.

Matsumoto, M. 2004, 'http://www.math.sci.hiroshima-u.ac.jp/ m-mat/eindex.html'.

Matsumoto, M. & Kurita, Y. 1992, 'Twisted gfsr generators', *ACM Trans. Model. Comput. Simul.* **2**(3), 179–194.

Matsumoto, M. & Kurita, Y. 1994, 'Twisted gfsr generators ii', *ACM Trans. Model. Comput. Simul.* **4**(3), 254–266.

Matsumoto, M. & Nishimura, T. 1998*a*, Dynamic creation of pseudorandom number generators, *in* H.Niederreiter & J.Spanier, eds, 'Monte Carlo and Quasi-Monte Carlo Methods', pp. 56–69.

Matsumoto, M. & Nishimura, T. 1998*b*, 'Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator', *ACM Trans. Model. Comput. Simul.* **8**(1), 3–30.

Panneton, F. & L'Ecuyer, P. 2005, 'Improved long-period generators based on linear recurrences modulo 2', *ACM Transactions on Mathematical Software - To Appear* .

pLab 2004, http://random.mat.sbg.ac.at/.

AUTHOR BIOGRAPHIES

MARCUS SCHOO holds a BSc (Mathematics and Computer Science) from the University of Canterbury, New Zealand. He is currently studying toward a BSc (Honours)(Computer Science), also at the University of Canterbury. His current research interests are in the area of pseudo-random number generators for parallel simulation and in the application of motion capture data in robotics.

DONALD C. MCNICKLE is an Associate Professor in the Management Department at the University of Canterbury. His research interests include queueing theory, networks of queues and statistical aspects of stochastic simulation. He is a member of INFORMS and the Operational Research Society.

KRZYSZTOF PAWLIKOWSKI is a Professor in Computer Science at the University of Canterbury, in Christchurch, New Zealand. The author of over 130 research papers and four books; he has given invited lectures at over 80 universities and research institutes worldwide. His research interests include performance modelling of telecommunication networks, discrete-event simulation and distributed processing. Senior Member of IEEE, member of ACM and SMSI.