Pseudo-Random Number Generators for Massively Parallel Discrete-Event Simulation

Adam Freeth[1], Krzysztof Pawlikowski[1], and Donald McNickle[2]

Departments of
[1]Computer Science and Software Engineering
[2]Management
University of Canterbury
Christchurch, New Zealand

December 16, 2011

Abstract

A significant problem faced by scientific investigation of complex modern systems is that credible simulation studies of such systems on single computers can frequently not be finished in a feasible time. Discrete-event simulation of dynamic stochastic systems, allowing *multiple replications in parallel* (MRIP) to speed up simulation time, has become one of the most popular paradigms of investigation in many areas of science and engineering. One of the general problems related with distributed simulation is the need of parallel generation of multiple sequences of pseudo-random numbers across cooperating processors, with the number of known, good parallel generators being very limited. This report assesses currently known techniques proposed for generation of pseudo-random numbers in processing systems, particularly the statistical properties of multiple sequences of numbers generated in parallel, and the speed of generation of these parallel streams and also the pseudo-random numbers themselves. Parallel implementations of the MRG32k3a and DX-120-2 generators are found to be the most suitable of those tested.

1 Introduction

Stochastic discrete-event simulation has become a crucial method for scientifically analysing the behaviour of telecommunication networks and other systems, particularly when actual experiments are difficult or impossible to undertake. The base for all models of these simulations is the underlying pseudo-random number generator (PRNG), which cannot be truly random due to the deterministic nature of computer logic. A good PRNG must therefore attempt to generate numbers that are indistinguishable from true random values, and be independent and identically distributed. Biases inherent in the generator affect the credibility of the final simulation results [33]. As PRNGs with good randomness properties are needed to be found, methods of testing the stochastic nature of these generators are necessary. Many such tests have been proposed in response to this [1, 11, 14, 15, 21, 23, 34], however all tests are only able to check for particular aspects of the randomness of a generator's output. Therefore, we require a thorough and comprehensive set of these tests to determine whether a given PRNG is able to produce numbers that cannot be differentiated from realisations of a probability distribution and can thus be guaranteed to give reliable simulation results.

Modern technologies and methodologies have given rise to the ability to implement complex simulation models for analysing detailed systems, which can take significant computational time to run. Since a large number of replications of simulation runs are usually required to achieve the required accuracy of observed performance measures, it can become extremely time-consuming to investigate these models. To alleviate this problem, the *multiple replications in parallel* (MRIP) paradigm has been proposed and is used in software packages such as Akaroa2[1] [7]. MRIP lets independent simulation runs to be run in parallel across a distributed computing platform, where observed data is sent from engines running the simulations back to a main controller to be analysed. This can significantly reduce the total time required to attain the sought-after results, provided that most of the computational time is spent on generating observation values.

To obtain uncorrelated simulation runs, each engine must be able to generate a unique and independent stream of pseudo-random numbers (PRNs), so they cannot merely use the same PRNG. Even if each engine was given a unique "random" seed to begin generating values from, there would remain a chance that the numbers generated by any two streams could start to overlap, particularly if the PRNG's period is small, as there could be no guarantee to the distance that the seeds are separated by in the cycle of the PRNG. Thus, parallel PRNGs (PPRNGs) – generators that are able to be parallelised across multiple processors into uncorrelated streams – are needed. Similarly with standard PRNGs, the individual streams of a PPRNG must generate numbers indistinguishable from a truly random sequence. The sequence must also be independent of that of any other stream, and a significant number of these streams need to be able to be created. Since the purpose of having MRIP is to speed up the evaluation of a simulation model, the time taken to initialise all streams must be minimal, so as not to trivialise this advantage of having MRIP.

In Section 2, we survey an array of modern test suites for the statistical analysis of PRNGs, including the range of tests and flexibility offered by each suite, and the ability to test for inter-stream correlations. We then look at the proposed methods of parallelising PRNGs in Section 3, and survey a variety of popular parallelisable generators in Section 4. This is followed by an experimental comparison of the generation speed and stochastic qualities of the generators in Section 5, with concluding remarks in Section 6.

2 PRNG Test Suites

We compare five popular statistical test suites for PRNGs to determine which one would most comprehensively assess the quality of a given PRNG, or if a combination of suites is required. In addition, we look for test suites capable of investigating correlations between streams in a parallel PRNG.

2.1 Diehard

The Diehard battery of tests, developed by Marsaglia [21], is a widely used battery that contains a number of statistical tests for PRNGs, implemented in Fortran and C. However, it offers no customisation for the sequence of tests it performs, nor the parameters of these tests, such as sample sizes. The implementation is also inconvenient in that it requires that the PRNs are 32-bit integers, which is often not the case, and that these numbers must be contained within a large

[1]Akaroa2 is a controller for distributed stochastic simulation developed by the Simulation Research Group at the University of Canterbury, Christchurch, New Zealand. It is aimed at improving the credibility of results of simulation studies using sequential analysis and MRIP. See `http://www.cosc.canterbury.ac.nz/research/RG/net_sim/simulation_group.html`.

binary file that is passed to the suite. It is no longer developed, and was last modified on 4^{th} April 1998.

2.2 Dieharder

Dieharder [1] is a random number test suite that is an extended implementation in C of the Diehard battery of tests, and includes some tests from the Statistical Test Suite developed by the National Institute of Standards and Technology (NIST), as well as some original tests. It is intended to be an all-encompassing test suite, and is designed to push test statistics to unambiguous success or failure through sequential analysis. It also attempts to give an idea of why a generator failed a particular test, where possible. This suite was last updated on 22^{nd} October 2009.

2.3 Statistical Test Suite

The NIST Statistical Test Suite implements 15 statistical tests for PRNGs aimed for use in cryptographic applications, as defined in [34] in the C language. Since we are interested in the randomness rather than the cryptographic properties of the PRNGs, the tests offered by this suite are not necessarily crucial for determining good quality PRNGs for use in stochastic simulation. It is still being developed, being last updated on 11^{th} August 2010.

2.4 Scalable Parallel Random Number Generators Library (SPRNG)

The SPRNG test suite [23, 24] implements a number of "parallel" versions of the popular tests described by Knuth [11], as well as the "inherently parallel" sum-of-independent-distributions test. As of Version 4.0, it is implemented in C++ and Fortran (with previous versions in C and Fortran). The tests are parallel in that they can interleave the observations from a number of streams in a round-robin way, creating a single stream of numbers to test upon. It also implements some physical model tests, based on the ISING model and random walk tests [22]. SPRNG was last updated on 7^{th} June 2007.

2.5 TestU01

TestU01 [18] contains a vast number of empirical tests for PRNGs implemented in C, divided into eleven modules by either similarity or authorship, as outlined in Table 1. The suite allows the selection of a variety of parameters for each test, and each test gives the *p-value* of the test of the null hypothesis, rather than just a simple pass or fail.

The suite also predefines a number of batteries of tests, including the battery `PseudoDIEHARD`, which implements most of the tests found in the original Diehard battery, but is not considered very stringent [18]; as well as a similarly equivalent battery of the NIST test suite. As with the SPRNG test suite, TestU01 offers the ability to interleave a number of streams of PRNs into one stream, which can subsequently be applied by any of the tests.

TestU01 is still being maintained, and was last updated on 18^{th} August 2009.

Table 1: TestU01 modules.

smultin	Tests that generate random points in a unit hypercube divided into equally-sized cubic cells, and test the frequency of the number of points that fall in each cell against the multinomial distribution.
sentrop	Tests that compute continuous and discrete empirical entropies of blocks of PRNs, and compares these to theoretical distributions.
snpair	Tests based on distances between closest points in a t-dimensional unit torus.
sknuth	Classical statistical tests described by Knuth [11], including the serial, permutation, gap, poker, coupon collector, run, maximum-of-t and collision tests. Some of these are merely special cases of the multinomial tests.
smarsa	Tests proposed by Marsaglia [20], including overlapping versions of the serial and collision tests, and the birthday spacings, binary matrix rank, Savir and GCD tests.
svaria	Tests that determine uniformity based on reasonably simple statistics, such as the mean and autocorrelation of a sequence of PRNs, and compare these to theoretical distributions.
swalk	Tests based on discrete random walks.
scomp	Tests based on linear complexity of a bit sequence as it grows, and a test for the Lempel-Ziv compressibility of bit sequences.
sspectral	Spectral tests that compute the discrete Fourier transform on strings of bits and test for divergences in consistency with the null hypothesis.
sstring	Tests applied to strings of random bits, including finding the correlations and longest sequences of ones in the strings, and the Hamming weights of strings.
sspacings	Tests based on the sum-functions of spacings between sorted observations [14].

2.6 Comparison of Test Suites

Table 2 below shows the tests implemented by each of the test suites. Implementation is shown by an "X".

Table 2: Comparison of statistical tests implemented by surveyed test suites.

Test	**TestU01**	**Diehard**	**Dieharder**	**STS**	**SPRNG**
Multinomial [18, p. 103]	X				
Multinomial (overlapping serial) [18, p. 104]	X				
Multinomial (bits) [18, p. 104]	X				
Multinomial (overlapping bits) [18, p. 104]	X	X[2]	X[3]	X[4]	
Discrete entropy [18, p. 105]	X				
Discrete entropy (overlapping) [18, p. 105-6]	X				
Entropy (Dudewicz and van der Meulen) [18, p. 106]	X				
Close pairs [18, p. 109]	X	X[5]	X[5]		
Close pairs (bit match) [18, p. 109]	X				
Bickel and Breiman statistic [18, p. 109]	X				
Serial [18, p. 110]	X				X
Permutation [18, p. 110-1]	X				X
Gap [18, p. 111]	X				X
Poker [18, p. 111]	X				X
Coupon collector [18, p. 111]	X				X
Run [18, p. 111]	X	X	X		X[6]
Maximum-of-t [18, p. 112]	X				X
Collision [18, p. 112]	X				X
Serial (overlapping) [18, p. 113]	X				
Collision (overlapping) [18, p. 113]	X	X[7]	X[7]		
CAT [18, p. 114]	X				
CAT (bits) [18, p. 114]	X			X	
Birthday spacings [18, p. 114-5]	X	X	X		
Binary matrix rank [18, p. 115]	X	X	X	X	

[2]As the Bitstream test [21], which is closely related with Delta $= -1$, $n = 2^{21}$, $L = 20$.

[3]As the Serial test [1], with Delta $= 1$; and as the Bitstream test (see above).

[4]As the Serial test [34], using Delta $= 1$; the Approximate Entropy test, using Delta $= 0$; and as the similar but weaker Overlapping Template Matching test.

[5]As the Minimum Distance test [21], being closely related using $N = 100$, $n = 8000$, $t = 2$, $p = 2$, $m = 1$; and the 3-D Spheres test, using $N = 20$, $n = 4000$, $t = 3$, $p = 2$, $m = 1$.

[6]Implements runs up only.

[7]As the Overlapping-Pairs-Sparse-Occupancy test [21], which corresponds to $n = 2^{21}$, $d = 1024$, $t = 2$, and $r = 0$ to 22; as the Overlapping-Quadruples-Sparse-Occupancy test. corresponding to $n = 2^{21}$, $d = 32$, $t = 4$, and $r = 0$ to 27; and as the DNA test, with $n = 2^{21}$, $d = 4$, $t = 10$, and $r = 0$ to 30.

Table 2 – continued from previous page

Test	TestU01	Diehard	Dieharder	STS	SPRNG
Savir [18, p. 115]	X	X[8]	X[8]		
GCD [18, p. 116]	X				
Sample mean [18, p. 117]	X				
Sample correlation [18, p. 117]	X				
Sample product [18, p. 117-8]	X				
Sum logarithms [18, p. 118]	X				
Weight distribution [18, p. 118]	X				
Collision arg max [18, p. 118-9]	X				
Sum collector [18, p. 119]	X				
Appearance spacings [18, p. 119]	X			X	
Random walk [18, p. 120-2]	X			X[9]	X
Linear complexity [18, p. 123-4]	X			X	
Lempel-Ziv compression [18, p. 124]	X			X	
Discrete Fourier transform [18, p. 125-6]	X			X	
Periods in strings [18, p. 127]	X				
Longest head run [18, p. 127]	X			X	
Hamming weight [18, p. 128]	X		X[10]	X	
Hamming weight correlation [18, p. 128]	X				
Hamming weight independence [18, p. 128-9]	X	X[11]	X[11]		
Run (bit) [18, p. 129]	X		X	X	
Autocorrelation [18, p. 130]	X				
Sum logarithms spacings [18, p. 131-2]	X				
Sum squares spacings [18, p. 132]	X				
Scan spacings [18, p. 132]	X				
All spacings [18, p. 132]	X				
Overlapping 5-permutation [21, cdoperm5.c]		X	X		
Parking lot [21, cdpark.c]		X	X		
Overlapping sums [21, cdosum.c]		X	X		
Craps [21, craptest.c]		X	X		
GCD (Brown) [1]			X		
Generalised minimum distance [1]			X		
Permutations (Brown) [1]			X		

[8] As the closely related Squeeze test [21].
[9] As the closely related Cumulative Sums, Random Excursions, and Random Excursions Variant tests [34].
[10] As the Monobit test, corresponding to $L = n$ [1].
[11] As the Count-the-1's test [21], a 5-dimensional overlapping version.

Table 2 – continued from previous page

Test	TestU01	Diehard	Dieharder	STS	SPRNG
Lagged sum [1]			X		
Kolmogorov-Smirnov test [1]			X		
Equidistribution [24, p. 453]					X
Sum of independent distributions [24, p. 455]					X
Metropolis algorithm [22]					X
Wolff algorithm [22]					X

Table 2 shows that the TestU01 suite clearly offers the greatest range of statistical tests for PRNGs, and includes most of the popular tests from the Diehard battery and those described by Knuth [11]. It also offers the ability to interleave parallel streams to test for correlations between them, with the only other test suite offering this being the SPRNG package. The only test of particular note not offered by TestU01 is the sum-of-independent-distributions test in the SPRNG suite. This test is inherently parallel, and the only test for correlations between parallel streams that goes beyond merely interleaving the streams, which all TestU01 tests are limited to.

TestU01 also provides a range of comprehensive batteries of tests that can be applied to particular PRNGs, making it easy to test the quality of a generator exhaustively, with standardised results allowing thorough comparison between generators [17].

TestU01 offers by far the widest variety of tests, with little else additional provided by other suites, as well as being easy to use and able test correlations between parallel streams, making it a substantial and comprehensive suite for testing parallel PRNGs. It had also been under development recently, keeping it up-to-date with modern statistical tests, and thus will be used for testing the stochastic qualities of PRNGs in this paper.

3 Parallelisation

Two predominant methods exist for parallelising a PRNG of a single stream $\{x_0, x_1, \ldots, x_{p-1}\}$ with period p to create multiple independent substreams: *cycle splitting* and *parameterisation.*

3.1 Cycle Splitting

Cycle splitting involves dividing the cycle of a large-period PRNG into P non-overlapping substreams, which can be achieved through either the *blocking* or *leap-frog* method. For either case, each substream's maximum length $\rho_i = \rho/P$, so the period of the initial generator must be very large if a great number of streams of appropriate length are required.

The **blocking method** has the i^{th} engine assigned the substream $\{x_{iB}, x_{iB+1}, \ldots, x_{iB+(B-1)}\}$ with block size B [35]. Generating all PRNs up to x_{iB} to give to the i^{th} engine would be a needlessly costly process, counteracting the advantages of generating numbers in parallel. Thus, an approach for jumping ahead to a given state is needed – such approaches are well-documented in literature [6, 8, 12]. The blocking method requires that the initial states of all streams must be generated sequentially, so the time for initialising the streams increases linearly with the number of streams needed. PRNGs must be chosen with minimal long-range correlations between generated numbers, as this can give short-range intra-stream correlations using the blocking method [25].

Using the **leap-frog method**, the i^{th} engine is assigned the substream $\{x_i, x_{i+P}, x_{i+2P}, \ldots\}$. This substream would produce numbers extraordinarily slowly if each stream x_i had to generate all consecutive numbers x_{i+jP} to $x_{i+(j+1)P}$ to get the $(j+1)^{th}$ value from the j^{th}. Therefore, it also requires the use a jump-ahead approach. As jumping ahead is likely to be significantly more computationally intensive than the PRNG itself generating successive PRNs, substreams implemented via leap-frogging will output successive values much more slowly than using blocking. PRNGs must again have minimal long-range correlations, as this can create short-range inter-stream correlations.

3.2 Parameterisation

Parameterisation is the process of producing independent full-cycle generators for each stream. This can be achieved by *seed parameterisation*, where initial seeds are varied to construct independent streams, or by *iteration function parameterisation*, where values in the PRN generation function are altered for each stream [35]. Careful parameter selection is required to ensure that each generator produces streams that have a full-cycle period and are independent of one another. Unlike cycle splitting, initialisation of all the streams in parameterisation is able to be accomplished in parallel, as each engine can be assigned a *stream ID* that is used by a predefined method on that engine to create a unique stream. Each type of PRNG will be limited in the number of possible unique and independent streams it can instantiate.

4 Generators

4.1 Mersenne Twister

The Mersenne Twister is a class of the twisted generalised feedback shift register (TGFSR) PRNGs introduced in Matsumoto and Nishimura [28], which uses Mersenne-prime periods and is based on the recurrence:

$$\vec{x}_{k+n} = \vec{x}_{k+m} \oplus (\vec{x}^u_k | \vec{x}^l_{k+1})A, \quad (k = 0, 1, \ldots) \tag{1}$$

$$\vec{z}_{k+n} = \vec{x}_{k+n}T \tag{2}$$

where $\vec{x}_i$ is the i^{th} word vector of size w, "$\oplus$" is bitwise `xor`, "$|$" is concatenation, and $\vec{x}^u_k$ is the upper $w-r$ bits and $\vec{x}^l_k$ is the lower r bits of $\vec{x}_k$ for a given integer r such that $0 \leq r \leq w-1$. A is the $w * \times w$ twisting matrix and T is the $w \times w$ tempering matrix. As the generator performs only bitwise operations without multiplication or division, it is very fast.

The most popular implementation of the Mersenne Twister is MT19937, which is equidistributed in up to 623 dimensions with 32-bit accuracy, and has a period of $2^{19,937}-1$. Although considered a generator with good randomness properties, passing almost all tests in the `BigCrush` battery of the TestU01 suite [17], it was found to be far from optimally equidistributed in large dimensions [32].[12] The generator performs a small amount of bit transformations at each recurrence, meaning that at states where very few of the bits are set to one, it can take hundreds of thousands of successive output vectors before this property is mitigated, i.e., there is a large period of poor randomness. Due to the huge period given by implementations such as MT19937, such states are extremely unlikely to arise, but still remain a possibility. Thus, when using a Mersenne Twister PRNG in simulations where a high degree of randomness is necessary, it must be recommended that the simulation be repeated at least twice with varying initial seeds to ensure that sequences with many zero-bits are not encountered.

[12]For the set ψ_t of all vectors of successive t values generated, a linear PRNG over $\mathbb{F}_2$ is optimally equidistributed in the i^{th} dimension when a unit hypercube $(0,1]^t$ is dissected into 2^l equal cells (for any positive integer l), and each cell contains exactly 2^{k-tl} of its points.

A number of methods for jumping ahead in generators based on linear recurrences modulo 2, such as the Mersenne Twister, exist [8, 9]. A naive method of jumping ahead is simply to multiply the generator's state-space by a precomputed $k \times k$ matrix $A := F^J$, where F is the matrix representation of the state transition function and J is the number of steps to jump ahead. However, since this matrix requires k^2 bits of memory and that k is generally very large ($k = 19,937$ for MT19937), it would require excessive memory usage and time-consuming matrix-vector multiplication.

For faster methods of jumping ahead, the coefficients of the polynomial $g(t)$ can be precomputed as:

$$g(t) := t^J \bmod \left(\varphi_F(t) = \sum_{i=0}^{k-1} a_i t^i \right) \tag{3}$$

where $\varphi_F(t)$ is the minimal polynomial of transition matrix F. Then, it is possible to jump ahead J steps from a given state-space s_0 using Homer's method [9]:

$$F^J s_0 = g(F) s_0 = F(\ldots F(F(a_{k-1}s_0) + a_{k-2}s_0) + \cdots) + a_0 s_0 \tag{4}$$

As the matrix-vector multiplication Fs is equivalent to stepping ahead to the subsequent state, which can be computed quickly due to the few bitwise operations required, Horner's method takes $O(k^2)$ time, with F applied k times, and about $k/2$ vector additions. The speed can be increased by applying the sliding window algorithm, whereby $h(F)s_0$ is precomputed for all polynomials $h(t)$ of degree no greater than a constant q, creating a table $2^q k$ bits in size. The value of k can be chosen to optimise the speed of jumping ahead. This decreases the number of vector additions to approximately $2^q + \lceil k/(q+1) \rceil$. The optimal value for enhancing speed for $k = 19,937$ is $q = 7$, requiring a precomputed table of about 312Kb in memory [9].

Mersenne Twister PRNGs are also able to be parallelised through parameterisation, using the method of Dynamic Creation introduced in Matsumoto and Nishimura [29]. In Dynamic Creation, each processor can be given the generator's specification, such as word size and period, as well as a unique ID. The processors each then construct unique and independent Mersenne Twisters, having irreducible and distinct characteristic polynomials of their recurring sequences, which is ensured by encoding their given IDs into the parameters of the generator.

4.2 WELL Generator

Panneton and L'Ecuyer [32] proposed a well equidistributed long-period linear (WELL) PRNG, another class of TGFSR generators, in response to the non-optimal equidistribution and inability to quickly escape periods of the state-space with a large fraction of one-bits as found in the Mersenne Twister, and a lack of maximally-equidistributed generators that can generate PRNs at similar speeds. Like the Mersenne Twister, the WELL generator performs only bitwise operations on the words in its state-space; however, it applies these operations to a greater number of the w-bit blocks at each step, and so is able to recover from a period of many one-bits much faster (approximately a thousand times faster for $k = 19,937$).

k can be dissected into $k = rw - p$, where r and p are distinct integers where $r > 0$ and $0 \leq p < w$, and the state vector $\vec{x}_i$ is divided into its w-bit blocks as $\vec{x}_i = (\vec{v}_{i,0}, \ldots, \vec{v}_{i,r-1})$. Linear transformations are applied to these blocks through the $w \times w$ binary matrices $T_0, \ldots, T_7$, and are chosen, along with integers k, p, m_1, m_2 and m_3 such that $0 < m_1, m_2, m_3 < r$, so the characteristic polynomial of the step transitional matrix A is primitive on $\mathbb{F}_2$. A bit mask that sets the last p bits of a block to zero is given by $\vec{m}_p$, bitwise `xor` is given by "$\oplus$" and left rotation by p bits is denoted by rot_p. $\vec{y}_i$ is the output vector. The algorithm for generating successive state vectors is detailed in Figure 1.

Due to the greater number of operations at each step, a WELL generator will be slower at generating PRNs than its equivalent Mersenne Twister. It is able to be parallelised in exactly the same ways as the Mersenne Twister, using both blocking and Dynamic Creation.

Figure 1: The WELL algorithm for generating a successive pseudo-random state vector $\vec{y}_i$.

```
$\vec{z}_0 := \mathrm{rot}_p(\vec{v}_{i,r-2}^T, \vec{v}_{i,r-1}^T)^T$
$\vec{z}_1 := T_0\vec{v}_{i,0} \oplus T_1\vec{v}_{i,m_1}$
$\vec{z}_2 := T_2\vec{v}_{i,m_2} \oplus T_3\vec{v}_{i,m_3}$
$\vec{z}_3 := \vec{z}_1 \oplus \vec{z}_2$
$\vec{z}_4 := T_4\vec{z}_0 \oplus T_5\vec{z}_1 \oplus T_6\vec{z}_2 \oplus T_7\vec{z}_3$
$\vec{v}_{i+1,r-1} := \vec{v}_{i,r-2}\vec{m}_p$
for $j = r-2, \ldots, 2,$ do
    $\vec{v}_{i+1,j} := \vec{v}_{i,j-1}$
$\vec{v}_{i+1,1} := \vec{z}_3$
$\vec{v}_{i+1,0} := \vec{z}_4$
$\vec{y}_i := \vec{v}_{i,0}$
```

4.3 Combined Multiple Recursive Generator

Multiple recursive generators (MRGs) [12] are linear generators of the form:

$$x_i = (a_1x_{i-1} + a_2x_{i-2} + \cdots + a_kx_{i-k}) \bmod m \tag{5}$$

Integers $a_1, \ldots, a_k$ must be selected such that $\sum_{i=1}^{k} a_i^2$ is large for the generator to perform well in spectral tests [35], so combined with the number of multiplications and additions required, it is inefficient at quickly generating PRNs. L'Ecuyer [13] proposed combined multiple recursive generators (CMRGs) in response to this which, for J MRGs and $j = 1, \ldots, J$, the j^{th} recurrence is defined as:

$$x_{j,i} = (a_{j,1}x_{j,i-1} + a_{j,2}x_{j,i-2} + \cdots + a_{j,k}x_{j,i-k}) \bmod m_j \tag{6}$$

where a uniformly distributed PRN u_i on the interval $[0, 1)$ can be obtained, using arbitrary integers $\delta_1, \ldots, \delta_j$ so that δ_j is relatively prime to m_j for each j, by:

$$z_i = \left(\sum_{j=1}^{J} \delta_j x_{j,i}\right) \bmod m_1 \tag{7}$$

$$u_i = z_i/m_1 \tag{8}$$

These CMRGs will not have the requirement of needing many large coefficients to pass spectral tests, so generating good PRNs with them should not require as many time-consuming multiplication and addition operations, and thus will be faster.

One such popular implementation of a CMRG that is found to perform well in statistical tests is MRG32k3a [15] of period $\rho \approx 2^{191}$, which has $J = 2$ MRGs of $k = 3$ terms each, with the parameters $a_{1,1} = 0$, $a_{1,2} = 1,403,580$, $a_{1,3} = -810,728$, $m_1 = 2^{32} - 209$, $a_{2,1} = 527,612$, $a_{2,2} = 0$, $a_{2,3} = -1,370,589$ and $m_2 = 2^{32} - 22,853$.

CMRGs are able to be parallelised through cycle splitting, as the j^{th} MRG has the state vector $s_{j,i} = \{x_{j,i}, x_{j,i+1}, \dots, x_{j,i+k-1}$ for each $j = 1, \dots, J$, which a transition matrix A_j can be multiplied by to generate the successive state:

$$s_{j,i+1} = (A_j s_{j,i}) \bmod m_j \tag{9}$$

We can then precompute A_j^n to jump ahead to a given state n steps ahead as follows:

$$s_{j,i+n} = (A_j^n s_{j,i}) \bmod m_j \tag{10}$$

No implementation of parallelising CMRGs through parameterisation has yet been proposed.[13]

4.4 DX Generator

Deng and Xu [3] introduced fast MRGs with huge periods and equidistribution properties over high dimensions, each described by the format DX-k-s, where k is the number of previous output values a generated PRN is dependent upon, and s is the number of non-zero coefficients, at approximately $k/(s-1)$ spaces apart. It is able to generate numbers quickly while maintaining good stochastic properties, as all of the non-zero coefficients are equal, i.e.,

$$\alpha_1 = \alpha_{\lfloor k/(s-1) \rfloor} = \alpha_{\lfloor 2k/(s-1) \rfloor} = \cdots = \alpha_{\lfloor (s-2)k/(s-1) \rfloor} = \alpha_k = B$$

where $\lfloor x \rfloor$ is the floor function, so it is fast as only one multiplication (of multiplier B) is needed. The generator thus takes the form:

$$x_i = B(x_{i-k} + x_{i-\lfloor \frac{(s-2)k}{s-1} \rfloor} + \cdots + x_{i-\lfloor \frac{k}{s-1} \rfloor} + x_{i-1}) \bmod m \tag{11}$$

for $i \geq k$, with maximum period $m^k - 1$. Such generators ave been found for up to $k = 10,007$, obtaining period lengths of about $10^{93,384}$ [2].

For the special case of $B = \pm 2^r \pm 2^w$ for integers r and w, bit shifting and addition can be used in place of multiplication to speed up generation even further [36]; such an implementation is given in [3].

As DX generators are MRGs, they can be parallelised through cycle splitting by jumping ahead using Equation 10, for a single MRG. However, the matrix A has dimensions $k \times k$, making it slow when k is large due to the great number of matrix-vector multiplications at each jump.

4.5 Multiplicative Lagged-Fibonacci Generator

Multiplicative lagged-Fibonacci generators (MLFGs) are given by the recurrence:

$$x_i = (x_{i-k} \times x_{i-l}) \bmod 2^b, \quad k < l \tag{12}$$

for positive integers k and l, having a maximum possible period of $\rho = 2^{b-3}(2^l - 1)$, i.e., the period of the generator can be increased by increasing the lag l, with the required memory increasing

[13]The SPRNG library offers a "Combined Multiple Recursive Generator" with parameterisation, but this is in fact a parameterised linear congruential generator combined with an invariable MRG.

proportionally. As the generator is multiplicative, and the product of an odd and an even number is necessarily an even number, the generator will always end up producing solely even numbers if it is initialised with any. Thus, it must be initialised with all odd number seeds $\{x_{i-1}, \ldots, x_{i-l}\}$ to avoid any initial transient phase, and the least significant bit of the output must be discarded, as this will always be 1.

Both parameterisation and cycle splitting techniques exist for parallelising MLFGs. They can be seed-parameterised to generate up to $2^{(b-3)(l-1)}$ independent streams [24], and can be cycle split through the blocking method in $O(l^2 \log BJ)$ computational time [25].

4.6 Inversive Congruential Generator

Inversive congruential generators (ICGs) were introduced by Eichenauer and Lehn [4] as an alternative to linear congruential generators (LCGs). ICGs do not have any lattice structure in the distribution of tuples of successive PRNs that are unavoidable with LCGs. They are given by the form:

$$x_i = \begin{cases} (a\overline{x_{i-1}} + b) \bmod p, & x_{i-1} \geq 1 \\ b, & x_{i-1} = 0 \end{cases}, \quad 0 \leq x_i \leq p, \quad i < 0, \tag{13}$$

with period of prime number p, where x_0 is a non-negative integer less than p, a and b are positive integers, and $\overline{x}$ is the modular multiplicative inverse of x modulo p. The inverse $\overline{x}$ can be calculated by the Euclidean algorithm at an average of approximately $n \approx 12 \log(2/\pi^2) \log p$ steps, making it about $12 \log(2/\pi^2) \log p + 1$ times slower than that of an equivalent LCG. Thus, it can only be used where the speed of the generation becomes negligible, either in simulations where this generation takes up only a tiny fraction of the computational time, or when computer processing speeds become fast enough (doubling approximately every two years according to Moore's law [30, 31]).

A variation on these, explicit inversive congruential generators (EICGs), was proposed in Eichenauer-Herrmann [5], of the form:

$$x_i = \overline{a(i_0 + i) + b} \bmod p, \quad i \geq 0 \tag{14}$$

for positive integer i_0, which displays even better lattice properties than the original ICG. The EICG lends itself to excellent cycle splitting properties [10], making it a good candidate for parallelisation.

5 Empirical Comparison

The experiments were performed on an Intel Core2 Duo CPU at 2.66GHz, using Linux version 2.6.27.19 and Fedora 10.

5.1 Generator Implementations

When used in practice, publicly available implementations of PRNGs are usually used rather than new implementations, as this saves programming time and is guaranteed to be correctly implemented. This paper will also use already-implemented versions of these generators, as well as their parallelisation implementations, with the exception of the DX generator. All implementations are in the C language and were required to generate PRNs that are uniformly distributed over the interval $[0, 1)$.

An implementation of the Mersenne Twister with $k = 521$ (i.e., MT521) using Dynamic Creation was taken from [26]. Cycle splitting via blocking implementations of MT19937, as well as the equivalent WELL generator [27], were used with the implementations' maximum possible number of steps to jump ahead $J = 2^{31} - 1$. The `RngStream` package [16] was used to parallelise the MRG32k3a generator by the blocking method, given streams of length 2^{127}. A DX-120-2 generator, implemented in [3] with the special property of $B = \pm 2^r \pm 2^w$ to increase generation speed using bitwise operations, was used with a precomputed jump-ahead matrix using $J = 2^{31} - 1$ to create streams via blocking. An MLFG from the SPRNG library [24] with default lag $l = 17$ was parameterised to obtain up to approximately 2^{1008} distinct streams, each having a period of about 2^{81}. An EICG implementation was used from the PRNG library [19], however, each EICG could only have a maximum period of $2^{31} - 1$, so three EICGs[14] were combined to get a Combined EICG (CEICG) with a period of about 2^{93}. This CEICG was then parameterised to obtain streams with a length of $2^{31} - 1$, an implementation maximum.

5.2 Stream Initialisation

Before each engine is able to begin generating PRNs, it must be initialised with the parameters (of the generator itself or of the initial seed) required to produce a stream of numbers distinct from any of the other streams. Initialisation time is considered as the time taken from when a machine requests the assignment of a stream of PRNs to the instant it is capable of generating the numbers themselves.

Using the blocking method, this means the controlling engine must compute the starting state of each stream sequentially, as it must jump ahead J steps to obtain this state for each new stream, and then this state is given to the appropriate engine to begin generating numbers. The time taken for a given engine to be able to begin generating PRNs, i.e., after all its appropriate parameters and its state have been computed, is measured, and then the average of these times for all engines is calculated. An average is taken rather than total initialisation time, as each individual stream is ready to begin generating PRNs as soon as it is initialised, regardless of whether the other streams are ready or not.

For parameterised PPRNGs, however, each engine can compute its own parameters once assigned a unique stream ID, and so initialisation of these streams is completely parallelised. Thus, for Dynamic Creation of the Mersenne Twister, the simulation engine created $1,000$ streams, and the average time taken for the initialisation of a single stream was calculated. The time taken for creation of new streams of the MLFG in the SPRNG library was dependent upon the number of previous streams that had been created, so it is not representative of parallelised initialisation, and thus was not measured. Schoo *et al.* [35], however, indicate that the stream initialisation of MLFG is many times faster than that of the Mersenne Twister when parallelised.

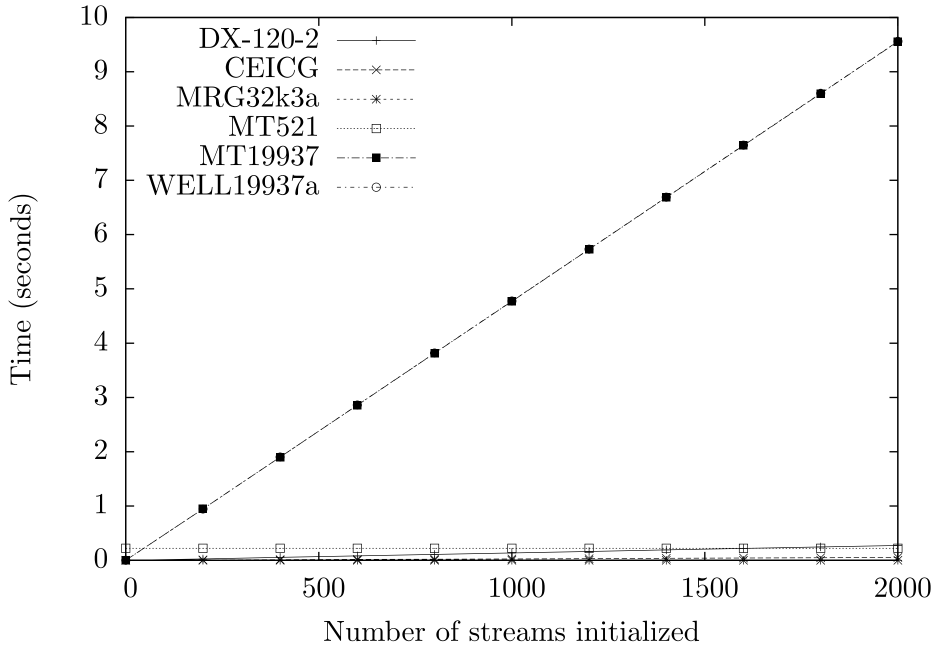
Table 3 details the times taken by the PPRNGs for generating massive numbers of streams, up to $60,000$. This is complemented Figure 2, which shows that the time taken for all PPRNGs using the blocking technique to initialise streams increases linearly with the number of streams, while the Mersenne Twister with Dynamic Creation takes constant time. The MT19937 and WELL19937a generators were clearly the slowest to initialise streams, both taking almost exactly the same amount of time due to the same method of jumping ahead. MRG32k3a was the fastest, being able to sequentially initialise 60,000 streams faster than the parallel Dynamic Creation of the MT521.

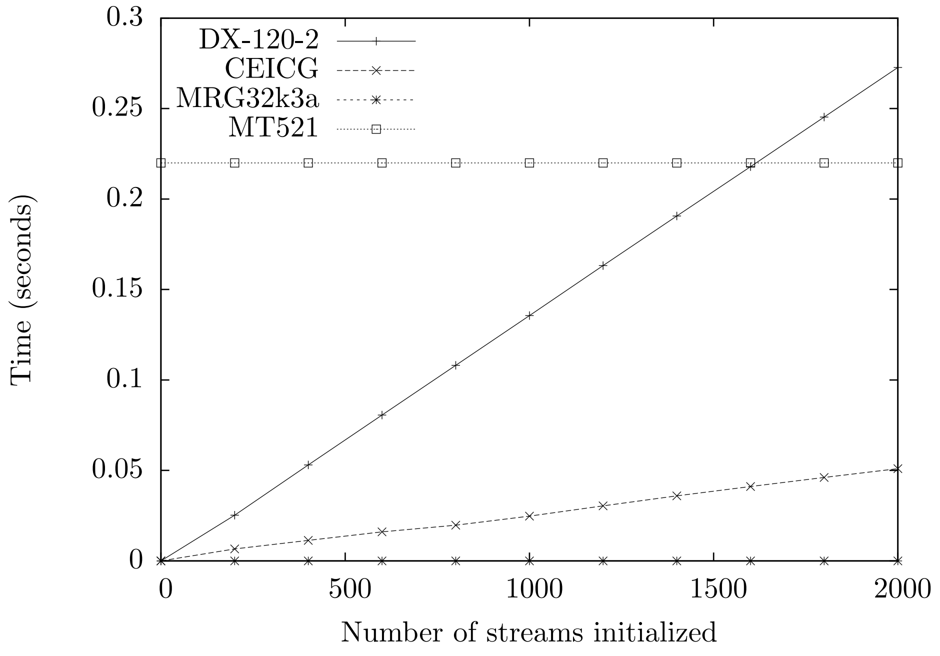
[14]Using the format EICG(p, a, b, i_0), the three EICGs are defined as EICG(2147483647, 7, 1, 0), EICG(2147483629, 11, 1, 0), and EICG(2147483587, 12, 1, 0).

Table 3: Average time taken in seconds for streams to be initialised for PRNGs, over a range of number of streams initialised. All generators parallelise with the blocking method unless otherwise stated.

Generator	**Number of streams initialised**					
	10,000	**20,000**	**30,000**	**40,000**	**50,000**	**60,000**
MT521 (Dynamic Creation)	0.220	0.220	0.220	0.220	0.220	0.220
MT19937	48.576	97.126	145.674	194.221	242.768	291.314
WELL19937a	48.609	97.193	145.770	194.341	242.909	291.477
MRG32k3a	0.004	0.013	0.020	0.030	0.038	0.046
DX-120-2	1.626	3.250	4.874	6.498	8.123	9.747
CEICG	0.222	0.439	0.657	0.875	1.092	1.310

Figure 2: Time taken for PPRNGs to initialise streams. All PPRNGs use the blocking technique for parallelisation, with the exception of MT521, which uses Dynamic Creation. *Note: the WELL19937a and MT19937 series overlap closely on the top plot.*





5.3 Single-Stream PRN Generation

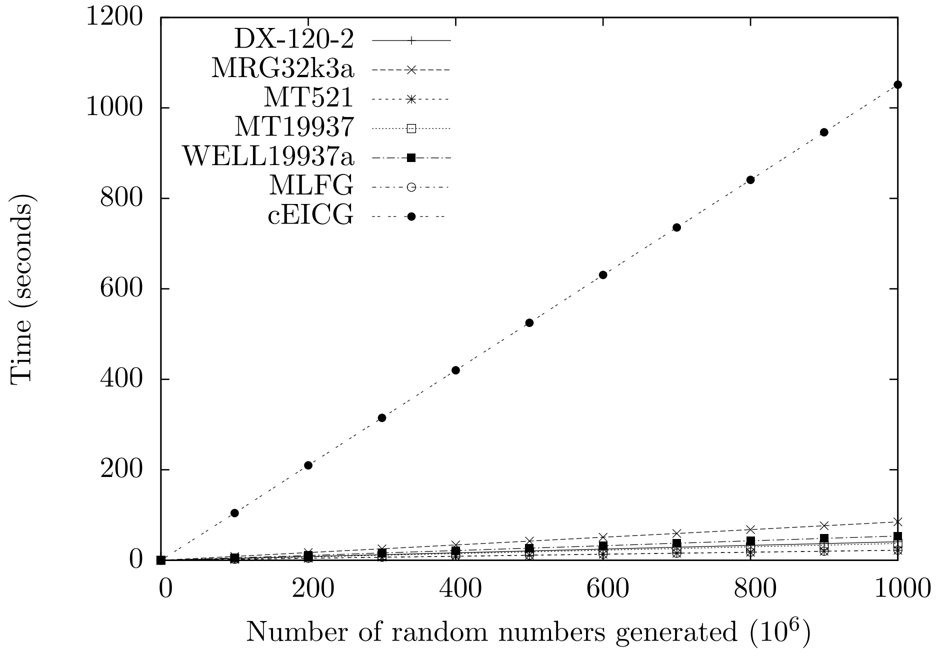
The stream of generation will be the same for each stream of a particular PPRNG, as the generator of each stream differs only in specific parameters or initial state. Thus, to measure the generation speed of a PPRNG, we only need to look at one of its streams.

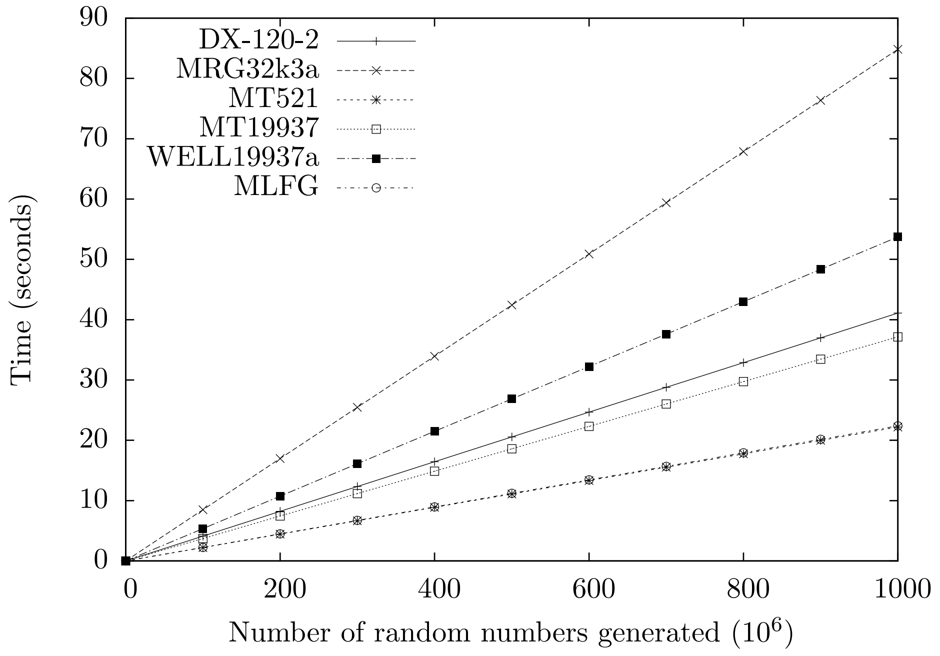
Table 4: Time taken for a single stream to generate 10^9 uniform PRNs over the interval $[0, 1)$, excluding any initialisation time.

PPRNG	**Time (s)**
MT521	22.20
MT19937	37.15
WELL19937a	53.72
MRG32k3a	84.83
DX-120-2	41.10
MLFG	22.40
CEICG	1,051.41

The MLFG and MT521 generators can be seen to generate PRNs the most quickly, at similar speeds to each other as seen in Figure 3. At each step, the MLFG only requires a single multiplication and the MT521 generator uses only bitwise operations over a smaller state-space compared to that of the MT19937 and WELL19937a generators, so they are faster than the other generators which perform more operations. The WELL19937a generator must compute a great number of bitwise operations to achieve better equidistribution properties than its equivalent Mersenne Twister, and thus performs slower. The DX-120-2 and MRG32k3a both perform multiple addition and multiplication operations, so are not as fast as generators such as the MLFG, with the MRG32k3a performing the worst and generating PRNs the slowest. In Table 4, the combined EICG can be seen to take a number of orders of magnitude longer to generate PRNs than all the other PPRNGs, having to compute an inverse in a large modulus at each step, and is thus excluded from the lower plots of Figures 3, 4 and 5 for clarity.

Figure 3: Time taken for single streams of PPRNGs to generate PRNs, excluding any initialisation time. The lower plot excludes the CEICG, which is far slower than the others. *Note: the MLFG and MT521 series overlap closely at the bottom of each plot.*





5.4 Parallelised PRN Generation

When comparing the time taken for parallel generators of large numbers of streams to generate PRNs, along with the generation time of the numbers themselves, we must also include the initialisation time needed to create the streams that generate these numbers. As seen in Section 5.2, different methods of parallelisation result in varying behaviours in the time taken for streams to be initialised.

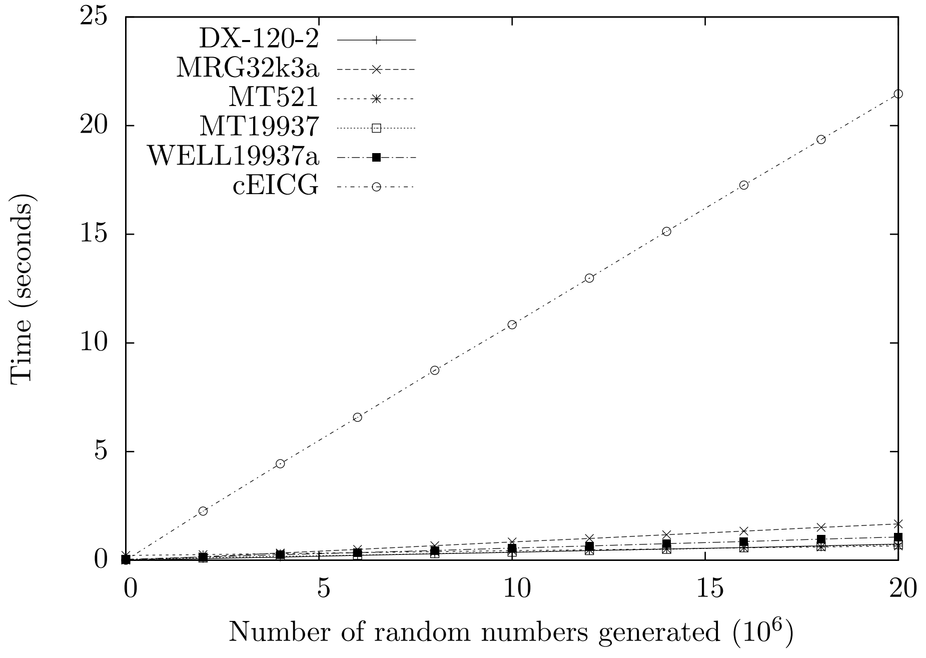
Figures 4 and 5 compare the PPRNGs in terms of the time taken for an average stream to generate PRNs, for a given number of streams. This includes the average initialisation time of these streams, which is given when the number of generated numbers is zero, i.e., where the plots intercept the y-axis. Thus, it is possible to analyse the magnitude of the effect of a PPRNG's initialisation time for varying numbers of PRNs to be generated.

The upper plots of both figures show that the fast initialisation of CEICG streams, as seen in Figure 2, is more than offset by its poor PRN generation speed—with it being the slowest PPRNG for each stream to generate one million PRN, when initialising either 10 or 100 streams. The CEICG performs even more unacceptably as greater numbers of generated numbers are required.

For the remaining PPRNGs, when generating numbers with 10 streams (see Figure 4), the initialisation times of these streams are largely insignificant. Generally, the ranked order of the total generation time of these PPRNGs is the same at small quantities of PRNs generated as at large quantities. The exception to this is the MT521 using Dynamic Creation, which has the longest initialisation time (being independent of the number of streams), but is able to generate PRNs the fastest. Thus, at a certain amount of PRNs required to be generated—approximately 16 million in each stream—it becomes the fastest generator even when we include the initialisation time of the 10 streams. Before this point, both the DX-120-2 and MT19937 generators are the quickest, with the latter having a slightly shallower gradient than the former, meaning it will be quicker in the long run.

As would be expected, the effect of initialisation becomes more considerable when 100 streams are needed, as seen in the lower plot of Figure 5. The effect of initialisation time on the DX-120-2 and MRG32k3a generators is seen to be negligible, whereas both TGFSR generators require the most time for initialisation. The DX-120-2 and MRG32k3a generators are, however, not the fastest at generating sequential PRNs, as given by the steeper slope of their plots. For 100 streams, the MRG32k3a can be seen to take the longest time for a stream to generate 20 million PRNs in each stream (with the exception of CEICG). DX-120-2 is faster, but is still slower than the MT521 generator at 20 million PRNs. In the long run, it can be seen that MT521 will generate the PRNs most quickly, followed by MT19937. It is obvious that this result is the same as for when we have just a single stream, because initialisation time required for any generator becomes negligible as the number of PRNs generated approaches infinity.

Figure 4: Average time taken by each stream of a PPRNG to generate PRNs, inclusive of initialisation time for 10 streams. The lower plot excludes the CEICG, which is far slower than the others.



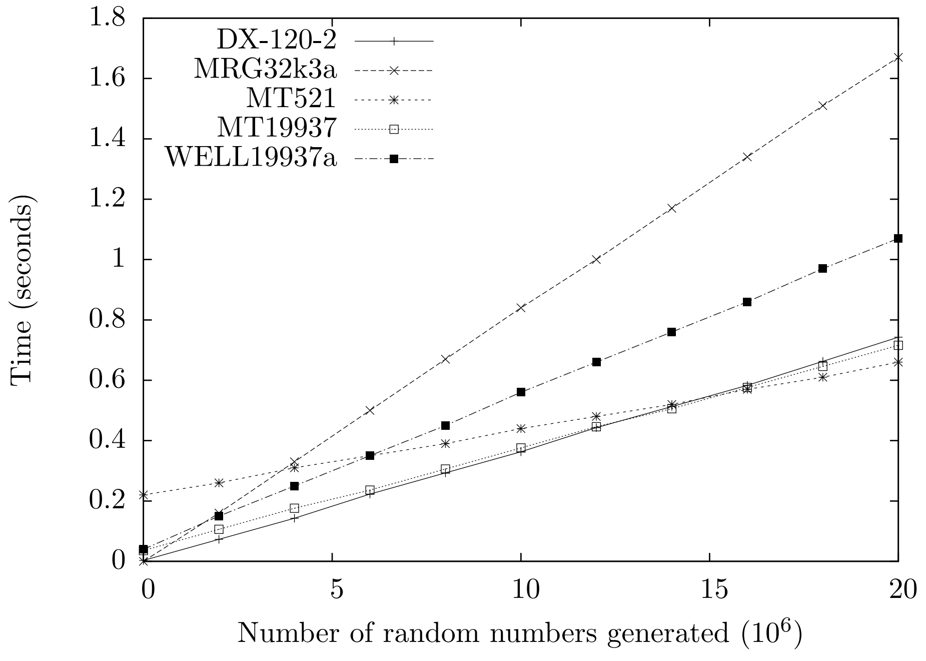
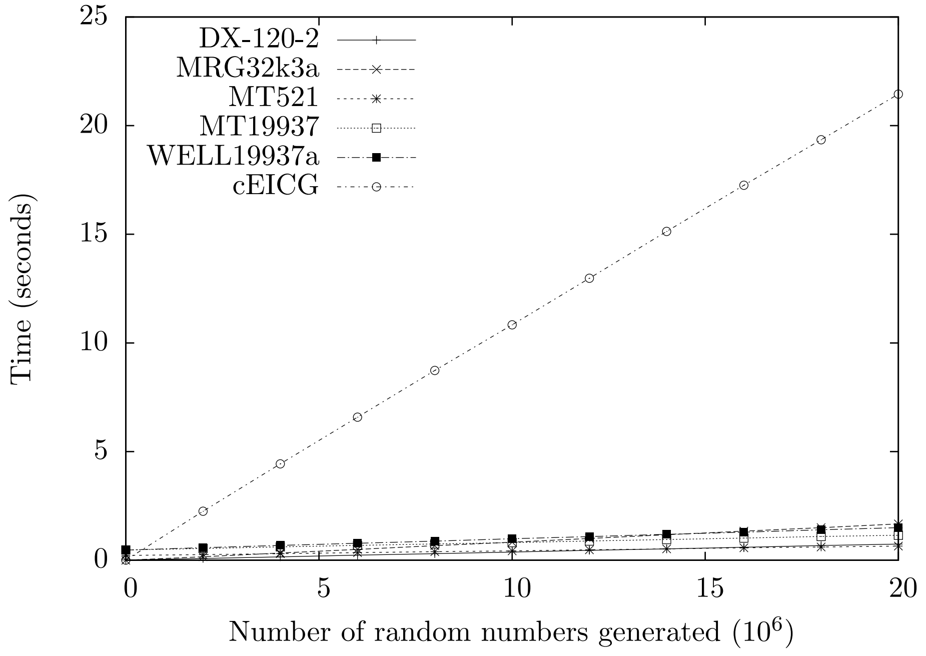
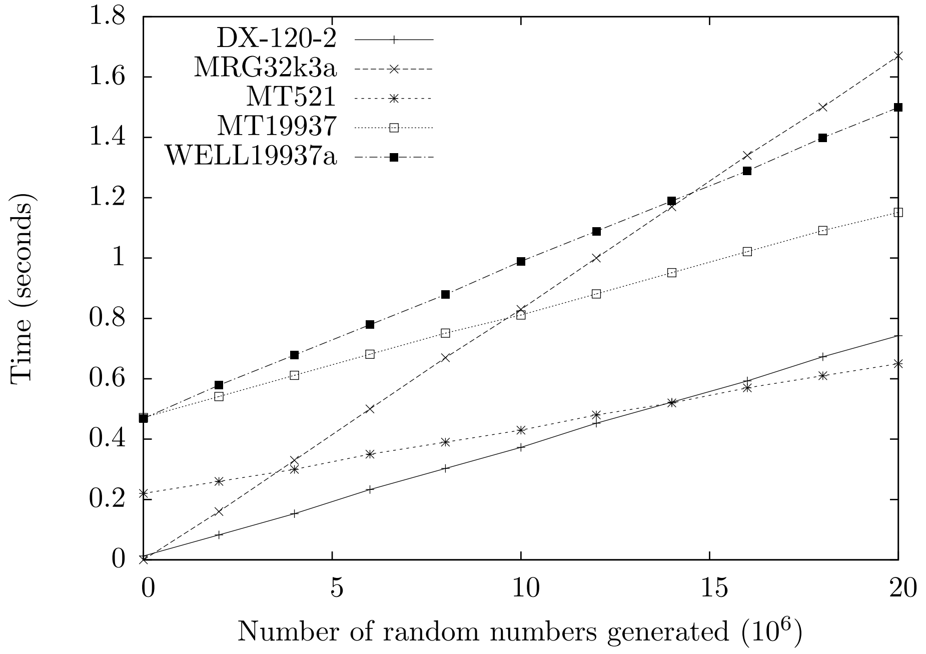


Figure 5: Average time taken by each stream of a PPRNG to generate PRNs, inclusive of initialisation time for 100 streams. The lower plot excludes the CEICG, which is far slower than the others.





5.5 Statistical Testing of Stochastic Properties

In Section 2, we found that the TestU01 suite was the most comprehensive set of tests for testing the quality of randomness of the output of PRNGs. L'Ecuyer and Simard [17] tested a wide variety of popular implementations of PRNGs on TestU01's `SmallCrush`, `Crush` and `BigCrush` batteries, the most stringent batteries TestU01 offers. However, they did not test some of the generators surveyed in this report, nor any parallelised implementations of generators.

Table 5 shows the results of both single-stream and parallelised versions of the generators, as tested on by the `SmallCrush`, `Crush` and `BigCrush` batteries. A number in the column of a battery indicates how many tests that particular PRNG clearly failed in that battery, i.e., where the p-value of that test is outside the interval $[10^{-10}, 1-10^{-10}]$. A number in parentheses indicates the number of questionable p-values obtained in the test, where the p-value lies within the previous interval but outside the interval $[10^{-4}, 1-10^{-4}]$. The available implementations for the parallel MT19937 and WELL19937a generators allowed only a maximum block size of $B = 2^{31} - 1$, giving streams that are unsuitable in length to be tested with the `Crush` or `BigCrush` tests (the former requiring approximately 2^{35} PRNs).

Table 5: Results of TestU01's batteries on single-stream and parallelised implementations of surveyed generators. Blanks indicate no failed p-values, while dashes indicate no data available.

PRNG	Parallel streams	SmallCrush	Crush	BigCrush
MT512	1		4 (1)	6
(Dynamic Creation)	10		2	2
	100			
MT19937	1		2	2
	10		–	–
	100		–	–
WELL19937a	1		2	2
	10		–	–
	100		–	–
MRG32k3a	1			
	10		(1)	
	100			
DX-120-2	1			
	10		(1)	
	100			
MLFG	1		3	3 (1)
	10			
	100	1		
CEICG	1			
	10			
	100			

The results in Table 5 show that, in terms of randomness properties, all generators tested are high-quality, with each failing fewer tests than the majority of those tested in [17]. This is even more pronounced for the parallelised generators, with none of those tested failing any tests in the batteries, with the exception of the MT512 with 10 streams and the MLFG with 100 streams (the latter likely being a statistical fluke). As the MT512 was the only parallel generator using the leap-frog method, this suggests that those generators parallelised via blocking methods do not show any significant long-range correlations between outputted numbers. Parallel versions of the TGFSR generators (MT19937 and WELL19937a) were not able to be tested, but the single-stream implementations failed less tests in both `Crush` and `BigCrush` than the MT512 generator, and are intrinsically more complex (having $k = 19,937$). Thus, it can be assumed that it is likely that these generators would similarly fail fewer tests when parallelised.

Of the single-stream versions of the generators, the MT512 generator performed the worst, although still only clearly failing six of the 106 `BigCrush` tests. The MLFG, MT19937 and WELL19937a generators each also failed some tests in both batteries, yet similarly little. The remaining generators—MRG32k3a, DX-120-2 and CEICG—each passed every test, whether parallelised or not. The MRG32k3a and DX-120-2 generators both gave one suspect result in the `Crush` tests when parallelised with 10 streams, but this is probably a statistical irregularity, as neither PRNG showed any suspect values for the `BigCrush` battery for any number of streams. As these three generators show no sufficient evidence of any poor randomness qualities under the most rigorous of available test batteries, they can be considered excellent PRNGs in terms of stochastic properties.

6 Conclusion

Massively parallel PRNGs for discrete-event simulation require fast initialisation of parallel streams and generation of PRNs, producing streams of numbers indistinguishable from truly PRNs, to ensure credible simulation results. This report surveyed a number of well-known test suites for evaluating this stochastic quality of PRNGs, and found that the TestU01 suite was the most comprehensive and usable. A range of popular PRNGs with known methods of parallelisation were also surveyed. The speed of stream initialisation and number generation of these were tested, and the randomness properties of these generators were tested using the TestU01 suite and compared, for both single-stream and parallelised implementations.

The MRG32k3a and DX-120-2 generators proved to be the most suitable for credible distributed simulation, both passing all tests in the most stringent batteries offered by the exhaustive TestU01 suite. MRG32k3a proved to initialise PRN streams extraordinarily faster than all other included generators, and while it generated numbers slower than most others, it still did so in reasonable time. On the other hand, the DX-120-2 was able to generate numbers more quickly than MRG32k3a, although being slower at initialising streams. The DX-120-2 also has the advantage of its simplicity of implementation, both the generator itself and its method of parallelisation, and it also can be easily extended to provide periods of incredible length.

Thus, as a basis for credible simulation results, the DX-120-2 generator is recommended, providing high-quality random number output and generating numbers at sufficient speeds, whether single-stream or massively parallel generators are required. However, if a large number of streams is required with each stream only needing to generate a small amount of PRNs, the MRG32k3a generator may be preferred.

Further research into variations of DX generators could help establish the implementation complexity appropriate for practical use, with less complex versions likely offering faster generation and stream initialisation speeds, and more complexity meaning higher stochastic quality and increased period lengths.

References

[1] R. Brown. Dieharder: A random number test suite, 2011.

[2] L.-Y. Deng, H. Li, and J.-J. H. Shiau. Scalable parallel multiple recursive generators of large order. *Parallel Computing*, 35(1):29 – 37, 2009.

[3] L.-Y. Deng and H. Xu. A system of high-dimensional, efficient, long-cycle and portable uniform random number generators. *ACM Trans. Model. Comput. Simul.*, 13:299–309, October 2003.

[4] J. Eichenauer and J. Lehn. A non-linear congruential pseudo random number generator. *Statistical Papers*, 27:315–326, 1986. 10.1007/BF02932576.

[5] J. Eichenauer-Herrmann. Statistical independence of a new class of inversive congruential pseudorandom numbers. *Mathematics of Computation*, 60(201):pp. 375–384, 1993.

[6] K. Entacher, A. Uhl, and S. Wegenkittl. Linear and inversive pseudorandom numbers for parallel and distributed simulation. In *Proceedings of the twelfth workshop on Parallel and distributed simulation*, PADS '98, pages 90–97, Washington, DC, USA, 1998. IEEE Computer Society.

[7] G. Ewing, D. McNickle, and K. Pawlikowski. Multiple replications in parallel: Distributed generation of data for speeding up quantative stochastic simulation. In *Proceedings of 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, Berlin, Germany, 1997.

[8] H. Haramoto, M. Matsumoto, and P. L'Ecuyer. A fast jump ahead algorithm for linear recurrences in a polynomial space. In *Proceedings of the 5th international conference on Sequences and Their Applications*, SETA '08, pages 290–298, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Ecuyer. Efficient jump ahead for $\mathbb{F}_2$-linear random number generators. *INFORMS J. on Computing*, 20:385–390, July 2008.

[10] P. Hellekalek. Inversive pseudorandom number generators: concepts, results and links. In *Proceedings of the 27th conference on Winter simulation*, WSC '95, pages 255–262, Washington, DC, USA, 1995. IEEE Computer Society.

[11] D. E. Knuth. *The art of computer programming, volume 2 (2nd ed.): seminumerical algorithms.* Addison-Wesley, Reading, MA, USA, 1981.

[12] P. L'Ecuyer. Random numbers for simulation. *Commun. ACM*, 33:85–97, October 1990.

[13] P. L'Ecuyer. Combined multiple recursive random number generators. *Operations Research*, 44(5):pp. 816–822, 1996.

[14] P. L'Ecuyer. Tests based on sum-functions of spacings for uniform random numbers. *Journal of Statistical Computation and Simulation*, 59(3):251–269, 1997.

[15] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):pp. 159–164, 1999.

[16] P. L'Ecuyer. Rngstreams: An object-oriented random-number package in c with many long streams and substreams, 2004.

[17] P. L'Ecuyer and R. Simard. Testu01: A c library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33, August 2007.

[18] P. L'Ecuyer and R. Simard. Testu01: User's guide, compact version, 2009.

[19] O. Lendl and J. Leydold. Prng user manual, 2006.

[20] G. Marsaglia. A current view of random number generators. In *Proceedings, Computer Science and Statistics: 16th Symposium on the Interface*, New York, 1985. Elsevier.

[21] G. Marsaglia. The marsaglia random number cdrom including the diehard battery of tests of randomness. Produced by Dept. of Statistics, The Florida State University under an NSF Grant, 1995.

[22] M. Mascagni and J. Ren. Sprng user's guide: Physical model tests, 2003.

[23] M. Mascagni and J. Ren. Sprng user's guide: Test suite, 2003.

[24] M. Mascagni and A. Srinivasan. Algorithm 806: Sprng: a scalable library for pseudorandom number generation. *ACM Trans. Math. Softw.*, 26:436–461, September 2000.

[25] M. Mascagni and A. Srinivasan. Parameterizing parallel multiplicative lagged-fibonacci generators. *Parallel Computing*, 30(7):899 – 916, 2004.

[26] M. Matsumoto. Dynamic creator home page, 2010.

[27] M. Matsumoto. Generating multiple disjoint streams of pseudorandom number sequences, 2010.

[28] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, January 1998.

[29] M. Matsumoto and T. Nishimura. The dynamic creation of pseudorandom number generators. In *Monte Carlo and Quasi-Monte Carlo Methods*, pages 56–69. Springer, 2000.

[30] G. Moore. Cramming more components onto integrated circuits. In *Electronics Magazine*. 1965.

[31] G. Moore. Excerpts from a conversation with gordon moore: Moore's law, 2005.

[32] F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Trans. Math. Softw.*, 32:1–16, March 2006.

[33] K. Pawlikowski, H. D. J. Jeong, and J. S. R. Lee. On credibility of simulation studies of telecommunication networks. *IEEE Communications Magazine*, 40(1):132–139, 2002.

[34] A. Rukhin, J. Soto, J. Nechvatal, E. Barker, S. Leigh, M. Levenson, D. Banks, A. Heckert, J. Dray, S. Vo, A. Rukhin, J. Soto, M. Smid, S. Leigh, M. Vangel, A. Heckert, J. Dray, and L. E. B. III. *A statistical test suite for random and pseudorandom number generators for cryptographic applications.* National Institute of Standards and Technology, Gaithersburg, Maryland, 2010.

[35] M. Schoo, K. Pawlikowski, and D. McNickle. A survey and empirical comparison of modern pseudo-random number generators for distributed stochastic simulations. Technical Report TR-CSSE 03/05, University of Canterbury, New Zealand, 2005.

[36] P.-C. Wu. Multiplicative, congruential random-number generators with multiplier $\pm 2^{k1} \pm 2^{k2}$ and modulus $2^p - 1$. *ACM Trans. Math. Softw.*, 23:255–265, June 1997.