Fast Generation of High-Quality Pseudorandom Numbers and Permutations Using MPI and OpenMP on the Cray XD1

Stephen Bique
Robert Rosenberg
Naval Research Laboratory
4555 Overlook Avenue, SW, Washington, D.C. 20375
stephen.bique@nrl.navy.mil, robert.rosenberg@nrl.navy.mil

ABSTRACT:

Random number generators are needed for many HPC applications such as real-time simulations. Users often prefer to write their own pseudorandom number generators. We describe useful techniques to find and implement fast, high-quality customized parallel pseudorandom number/permutation generators. We prove and demonstrate a computational advantage. We discuss how to choose moduli. We introduce an algorithm to find many multipliers based on LLL reduction and present a fast implementation. We propose an empirical method to select multipliers from a set of multipliers found using LLL reduction. We also introduce an algorithm to assign numbers to a permutation for statistical analysis. We present results of runs on the Cray XD1.

KEYWORDS:

Lehmer LCG, MCG, primitive root, Rabin-Miller Probabilistic Primality Algorithm, spectral test, LLL reduction

Introduction

A linear congruential generator (**LCG**) of the form

$$x_{n+1} = (\alpha x_n + c) \bmod m, \quad n \geq 0, \tag{1}$$

is an amazingly simple device introduced by Lehmer in 1949 to generate a pseudorandom sequence x_0, x_1, x_2, …, with $x_n \in \{0,1,2,\ldots, m-1\}$, which exhibits many of the statistical properties that characterize a random sequence, provided the integer constants α, c and m are chosen carefully with $m > 0$, $0 \leq \alpha < m$, $0 \leq c < m$, and $0 \leq x_0 < m$ [1]. While there exist plenty of good choices for these constants, there is no such notion as a perfect LCG and every LCG will fail some test. A compelling advantage of LCGs is the fact that experiments (simulations) can be repeated by using the same "seed" x_0 (a true random number generator presumably based on some physical phenomenon is not reproducible). **LCG**s have been so successful on sequential computers, it should not be surprising they are also successfully applied on parallel computers despite the inherently sequential nature of the computations.

A special case of the LCG is a multiplicative congruential generator (**MCG**) with "increment" c =0:

$$x_{n+1} = \alpha \cdot x_n \bmod m \tag{2}$$

Both the "multiplier" α and the seed x_0 for the MCG must be nonzero since otherwise $x_n = 0$ for all n. Eventually the LCG must produce an element that has occurred earlier in the sequence and then the sequence repeats, which determines the *period* of the LCG. Evidently a disadvantage of using an **MCG** is the period can never be equal to the "modulus" m, which is possible for an **LCG** but insignificant when m is large (one less element). The **MCG** offers a computational advantage compared to an **LCG** because there is one less operation (addition). Moreover, we shall see in the next section there is a way to exploit the modulo operation of a **MCG**.

Avoid choosing a small modulus m whenever a large number of random numbers are needed (because the period is always bounded by m). The largest period possible for an **MCG** can be achieved provided

the multiplier is a “primitive root” whenever the modulus is a prime number because Euler's φ(*m*), the number of integers between 1 and *m* and relatively prime to *m*, is exactly the *m* − 1 integers less than *m*. This fact from elementary number theory is next briefly explained. Given an integer *m* , the “order” of any integer α is the smallest integer *n* such that

$$\alpha^n \bmod m = 1. \tag{3}$$

A primitive root of a prime modulus m has order $n = m - 1$, i.e., all of the integers α, α^2, α^3, …, α^{m-1} are distinct (α generates all of the elements under multiplication). To see these elements must all be distinct, suppose by way of contradiction $\alpha^i \bmod m = \alpha^j \bmod m$ with $0 \le i < j < m - 1$. Since gcd(α,*m*)=1, there must exist a modular multiplicative inverse α^{-i} such that $(\alpha^i \cdot \alpha^{-i}) \bmod m = 1$. But then $1 \equiv \alpha^i \cdot \alpha^{-i} \equiv \alpha^j \cdot \alpha^{-i} \equiv \alpha^{j-i} \pmod m$, which contradicts the fact that α is a primitive root. Luckily, it is not necessary to exhaustively test every power *n* = 2,3, …, m-2, when *m* is large to determine if a given integer is a primitive root of a prime modulus. It suffices to test

$$\alpha^{(m-1)/f} \bmod m \neq 1 \tag{4}$$

for all primes *f* dividing *m* − 1. To verify this claim, we need the following identity, which holds when the binary operation ⊗ is addition, subtraction or multiplication (this is what makes modular arithmetic "nice"):

$$(\alpha \otimes \beta) \bmod m = ((\alpha \bmod m) \otimes (\beta \bmod m)) \bmod m \tag{5}$$

Assume the order on an integer α (mod *m*) is the integer *n*. Suppose $\alpha^p \bmod m = 1$. Write

$$p = q \cdot n + r, \quad 0 \le r < n. \tag{6}$$

Then

$$\begin{aligned} 1 &= \alpha^p \bmod m && \text{by supposition} \\ &= \alpha^{q \cdot n + r} \bmod m && \text{by (6)} \\ &= (\alpha^{q \cdot n} \cdot \alpha^r) \bmod m \\ &= \alpha^r \bmod m && \text{by (5)} \end{aligned} \tag{7}$$

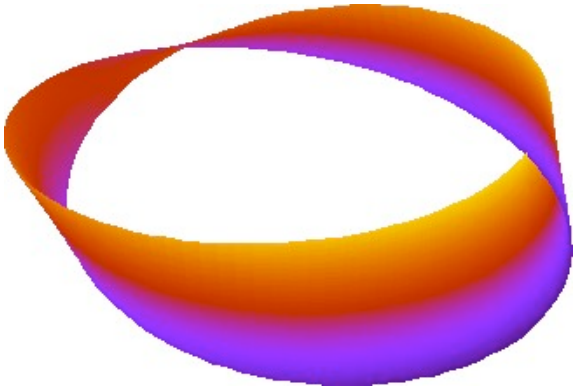
By assumption, *n* is the order of α and so equation (7) in view of the inequality (6) forces *r* = 0. Hence, the order of α necessarily divides *m−1* by Fermat's little theorem (1640), which states that the prime integer *m* divides $\alpha^{m-1}-1$ for all positive integers α that are relatively prime to *m*, i.e., $\alpha^{m-1} \bmod m = 1$. For each factor *f* of *m*−1, let ψ(*f*) denote the number of integers with order *f* (mod *m*). Gauss (1801), who was a wizard of summations already as a pupil in school based on a popular anecdote [2], showed that

$$\Sigma\psi(f) = m-1 = \Sigma\varphi(f). \tag{8}$$

and concluded that ψ(*f*) = φ(*f*) [3, p. 182], which completes the justification of the claim. Indeed, if α satisfies equation (3), then we have shown that the order of α divides *n*. Conversely, if α has order *r* and *r* divides *n*, then necessarily α satisfies equation (3). Thus, the number of solutions to equation (3) must be $\Sigma_{f|n}\psi(f)$, which Lagrange (1777) proved must be equal to *n* [4, p. 181]. Also $\Sigma_{f|n}\varphi(f) = n$ since there is a total of *n* terms

$$\frac{1}{n}, \quad \frac{2}{n}, \quad \frac{3}{n}, \quad \ldots, \quad \frac{n-1}{n}, \quad \frac{n}{n},$$

of which exactly φ(*f*) will have *f* in their denominator, after dividing out all common factors. Interestingly, Gauss' proof recovers ψ(*f*) = φ(*f*) from (8), which seems to require the Möbius inversion formula which was invented over half a century later in 1858 by the mathematician who invented the möbius strip [4][5].



A Möbius Strip

Usually it is easy to find small primitive roots regardless of the magnitude of the prime modulus. So it is practical to search for a primitive root starting from small primes. For example, using both the GNU Multiple Precision (MP) "Bignum" arithmetic (GMP) and the NTL libraries (with GMP) [6][7], we found that 11 is a primitive root of the prime $48047305725 \cdot 2^{172404} - 1 = 2864451626157029498 17\ldots$(16 pages worth of digits omitted)… 30342041599, which has exactly 51, 910 decimal digits (we counted using the UNIX command wc). This number is prime because $48047305725 \cdot 2^{172403}-1$ (which we confirm has the same number of digits base 10) is currently the largest known Sophie Germain prime (with high probability), which is any prime number p for which 2p+1 is also prime [8]. Such primes simplify the task of finding primitive roots. To show that a small number α is a primitive root of a large prime $2p+1$ where p is prime, it is only necessary to verify $\alpha^p \neq 1$. The reason is there are only two exponents that need to be checked by equation (4), namely $2p/p = 2$ and $2p/2 = p$. Plainly $\alpha^2 \bmod (2p+1) \neq 1$ since for small primes α, $\alpha^2 << 2p+1$.

Is there a sound approach to find constants for an **MCG** in a parallel setting using Sophie Germain primes? Search for a primitive root r of a prime modulus $m = 2p+1$, where p is a Sophie Germain prime, and then employ r^t as the multiplier for some suitable power t of the primitive root r [9]. Technically it is incorrect to say "Sophie-Germain primes as the prime modulus" [9] because p is the Sophie Germain prime, not $m = 2p+1$. For example, in [9] it is erroneously stated that $m = 2^{64} - 21017$ is a Sophie Germain prime ($2m + 1 = 2^{65} - 42033 = 36893488147419061199 = 23 \cdot 383 \cdot 42013 \cdot 253637 \cdot 393031$ is evidently not prime), although $p = 9223372036854765299$ is a Sophie Germain prime with $2p + 1 = 2^{64} - 21017$. Mersenne primes provide slightly better performance compared to using Sophie Germain primes but there are far fewer of them [9][10]. While there is a sound basis for using primes near a power of two as the modulus, the case for using Sophie Germain primes in not compelling.

A primitive root is discovered fairly quickly. Yet a small primitive root is unlikely a good multiplier for an **MCG** using a large modulus m. Is $r^n \bmod m$ a primitive root whenever r is a primitive root of a prime number? The powers of $r \pmod m$ generate all of the integers 1, 2, …, $m - 1$, and they cannot all be primitive roots because some exponents must divide $m-1$, which is even. All of the primitive roots belong to the set

$$\{ r^n \mid \gcd(m-1,n) = 1 \}. \tag{9}$$

Thus the number of primitive roots is the number of positive integers less than the prime modulus m that are not multiples of factors of $m-1$, which was discovered in the 19th century by Keferstein [3]. Regardless of the choice of the prime modulus and corresponding primitive root as a multiplier, terms of an **MCG** sequence with shift half of the period are highly correlated [11]. This fact favors employing large moduli m so that half the period $(m-1)/2$ is sufficiently large.

Factoring large integers using trial and error is practical only about 90% of the time. The reason is that given any sequence of 100 integers, most of them are divisible by 2 (50%), 3 (33%), 5 (20%), and other

small primes. Hence, not all integers are equally easy to factor. The **SQU**are **FO**rms **F**actorization algorithm can factor almost any integer with up to 18 digits [12][13]. Yet the algorithm may be useful when the word size is 128-bit or larger. Unlike the vast majority of factoring algorithms, the SQUFOF algorithm works best using mainly integer arithmetic.

We developed a fast variation of the SQUFOF algorithm. Our implementation in C is successful around 97% of the time for large integers up to 2^{64} (higher success rate is possible with reduced performance). While finding a factor may not be a trivial task for large integers, determining if any given integer has a non-trivial factor (i.e., is composite) can be decided quickly with high probability using the Rabin-Miller Probabilistic Algorithm, which was invented in the mid-1970's [14]. When the algorithm is correctly implemented, a program that asserts an integer is composite cannot be wrong. Perhaps a single composite among about 10^{20} integers will be indistinguishable from a prime number using the Rabin-Miller Probabilistic Algorithm.

Even when the number of factors is small and can be discovered reasonably fast, raising an integer to a large power via modular arithmetic is accomplished quickly and precisely but cannot be done using typical library functions. Identity (5) is useful to perform such computations. This identity is not typically covered extensively in computer science curricula. We developed a fast implementation in C to do such computations on unsigned integers. An implementation in C to perform modular multiplication on signed integers is published [15].

The popularity of **MCG**s has spawned an abundance of research in both sequential and parallel settings. Recently, a new approach to find optimal multipliers was proposed based on the “figures of merit”, a measure that was introduced in 1976 [16][17]. Generally it is impractical to conduct an exhaustive search to find optimal parameters suitable for sequential computations. In a parallel setting, there is no evidence to suggest that there exists any method of pseudorandom number generation that would be ideal for all applications. Regardless, packages for scalable pseudorandom number generation do exist [18].

Although a large body of knowledge exists on choosing parameters for an **MCG**, the theory is incomplete. For example, a statement such as “The multiplier is chosen to be certain powers of 13 … that give maximal period sequences of acceptable quality” [9] is sound, yet does not prescribe an algorithm. A programmer developing an application would find a table of parameters for an **MCG** useful. Yet, it is not easy to find a single multiplier for an **MCG** in many published works (see for example [9] or [10]). Moreover, practical issues are often inadequately addressed. For example, the period may be considerably smaller than theoretically possible due to computer arithmetic so that the effective period is $p^{½}$, where p denotes the theoretical period [19].

Our primary interest is to compute pseudorandom numbers in parallel while avoiding the computation of any subsequence produced by another process. Whenever two processes produce the same subsequence, we say that “overlap” has occurred. A natural idea is to use a different seed but to be certain that overlap is avoided, it is advisable to implement some “leapfrog” technique to partition the sequence among the processes, which may be appropriate if scalability is not a concern [20]. Although the leapfrog method is sound, we prefer a simpler approach that does not require additional computations whenever the number of processes is changed.

We seek a simple, fast implementation of an **MCG** which not only produces good results in a parallel setting for many applications, but also is easily adapted to run on different sets of processors, i.e., scalable in the sense that performance and quality remain high while the effort required to modify the code is minimal. In the next section, we focus on performance. Subsequently we prescribe ways to find prime moduli and good multipliers for an **MCG**. We present data based on our algorithms and then conduct some experiments to assess the quality of the results.

Fast Computation

If the computations are performed exactly, then it is possible to achieve the optimal period for the right choice of parameters. However, if overflow occurs during the computations, then the period will likely be

considerably smaller than predicted. We focus on relatively fast implementations, which limits the investigation as we shall see shortly.

When the modulus is a power of two, there is a computational advantage in performing the modulus operation for an **MCG** (using a fast bit-wise and operation). There is also a performance benefit when the modulus is a Mersenne or Sophie Germain prime [9][10]. Actually this advantage holds more generally for any prime sufficiently close to a power of two, which we demonstrate next.

For an **MCG** (2), fix $m = 2^q - k$ for arbitrary positive integers $q > 1$, $k \geq 1$. Write

$$\alpha \cdot x_n = \gamma\, 2^q + \lambda, \tag{10}$$

so that λ and γ hold the q least significant bits and the most significant bits, respectively, of the product $\alpha \cdot x_{n-1}$, which is the multiplier times the "previous" pseudorandom number in the sequence (2). Note $0 \leq \lambda \leq 2^q - 1$. Using the mathematician's usual addition and subtraction trick, we see that

$$\gamma\, 2^q + \lambda = \gamma\, 2^q + (k\gamma - k\gamma) + \lambda = \gamma\,(2^q - k) + k\gamma + \lambda. \tag{11}$$

Applying the nice identity (5) to the above equation (11), we immediately recognize that

$$x_{n+1} = \alpha \cdot x_n \bmod m = (k\gamma + \lambda) \bmod m. \tag{12}$$

Plainly replacing a single operation ($\alpha \cdot x_n$) by two operations ($k\gamma + \lambda$) is not what improves performance!

The mod m operation is faster using the right-hand side of equation (12) because (eventually) this operation can be accomplished using subtraction instead of an expensive division operation. We may have that $k\gamma + \lambda > 2m$, which means additional operations may be required to perform the mod m operation.

Example

Choose the prime modulus $m = 13$ with $q = 4$ and $k = 3$. Take $\alpha = 11$ which is a primitive root of 13. Set $x_0 = 12$, which is a valid seed for an **MCG**. Then $\alpha x_0 = 11 \cdot 12 = 132 = 8 \cdot 2^4 + 4$, which yields $\gamma = 8$ and $\lambda = 4$. Hence, $k\gamma + \lambda = 3 \cdot 8 + 4 = 28 > 26 = 2m$.

How large can γ be? For sufficiently small k, we claim

$$\gamma < 2^q - 2(k+1) \ . \tag{13}$$

Whenever α is a primitive root, we must have that $\alpha < m - 1$ because $m - 1$ is even whenever m is prime. We always have that $x_n \leq m-1$. Hence, $\alpha \cdot x_n < (m-1)^2$ for all integers x_n generated by an **MCG** (2). Therefore,

$$\alpha \cdot x_n < (m-1)^2 = (2^q - k - 1)^2 = 2^{2q} - 2^{q+1}(k+1) + (k+1)^2. \tag{14}$$

We know every nonnegative integer can be expressed as $2^q - k$ for some integers q and k. In particular, for every prime integer m with $2 < m < 2^q$, if m is sufficiently close to 2^q, then there exists a small integer k such that $m = 2^q - k$. We shall only consider integers k satisfying

$$1 \leq k < 2^{(q-1)/2}. \tag{15}$$

For such small integers k, we see that

$$(k+1)^2 \leq \left(\ 2^{(q-1)/2}\ \right)^2 \leq 2^{q-1} < 2^q. \tag{16}$$

In view of this inequality (16), upon dividing both sides of the inequality (14) by 2^q, we see that the claim (13) holds:

$$\gamma = \frac{\alpha \cdot x_n}{2^q} < \frac{2^{2q} - 2^{q+1}(k+1) + (k+1)^2}{2^q} = 2^q - 2(k+1) \ .$$

In the case of a Mersenne prime $m = 2^q - 1$, we need perform at most a single subtraction operation to perform the mod m operation in equation (12). Indeed, $k = 1$ and in view of the claim (13), we must have that

$$k\gamma + \lambda - m = \gamma + \lambda - m < 2^q - 4 + 2^q - 1 - m = 2^{q+1} - 5 - m = 2^{q+1} - 5 - (2^q - 1) = 2^q - 4 < m. \quad (17)$$

Example

Consider the miniscule Mersenne prime modulus $m = 7$ for illustration purposes. Choose the primitive root $\alpha = 5$. Set the seed $x_0 = 5$. Then $\gamma = 5 \cdot 5 / 8 = 3 < 2^q - 2(1+1) = 4$. This shows the upper bound predicted by (13) is tight. The sequence generated by the **MCG** (2) is 5,4,6,2,3,1, …, is calculated using $\gamma + \lambda$ [− m if needed] as follows: 3 + 1 = 4, 2 + 4 = 6, 3 + 6 − 7 = 2, 1 + 2 = 3, 1 + 7 − 7 = 1. Observe one third of the time subtraction is needed and the largest $\gamma+\lambda-m$ value is 2 (compute x_3 given $x_2 = 6$) and this maximum value is smaller than $2^3 - 4 = 4$ as the inequality (17) asserts.

Algorithm 1: MCG Computation for Mersenne Primes

Input: An integer x with $1 \le x \le m - 1$, a multiplier α with $1 \le \alpha < m$, a positive integer q with $2^q - 1 = m$
Output: $\alpha \cdot x \bmod m$.

1. $x \leftarrow \alpha \cdot x$
2. $x \leftarrow \gamma + \lambda$ where x has the binary representation $[\gamma \mid \lambda]$ with λ holding the q LSB of x
3. If $x > m$ then $x \leftarrow x - m$

Recall, we may have that $k\gamma + \lambda > 2m$. How large can $k\gamma + \lambda$ be if m is not a Mersenne prime? Consider a recursive application of equation (12) replacing $\alpha \cdot x_n$ by $k\gamma + \lambda$ to obtain new upper γ' and lower λ' "q-bits." We compute

$$k\gamma' + \lambda' = k \cdot \frac{k\gamma + \lambda}{2^q} + \lambda' \le k \cdot \frac{k(2^q - 2(k+1) - 1) + (2^q - 1)}{2^q} + \lambda'$$

$$= k \cdot \frac{2^q(k+1) - 2k(k+1) - (k+1)}{2^q} + \lambda' \le k(k+1) + \lambda' \le 2^{q-1} + 2^q - 1 = 3 \cdot 2^{q-1} - 1,$$

where the last inequalities hold by dropping the negative terms and noting that $k(k+1) \le 2^{(q-1)/2}(2^{(q-1)/2}) \le 2^{q-1}$ by inequality (15). Thus,

$$k\gamma' + \lambda' - m < 3 \cdot 2^{q-1} - 1 - (2^q - k) = 2^{q-1} + k - 1 < m, \quad (19)$$

where the last inequality must hold because k is small. To see this statement is true, note there are $2^q - 2^{q-1} = 2^{q-1}$ integers between 2^{q-1} and 2^q and the distance from m to 2^q is $|2^q - m| = k$, which is no more than half the distance by inequality (15). These calculations prove that at most two recursive applications of equation (12) are sufficient to compute the next pseudorandom number in (2).

Example

Consider the prime modulus $m = 1021$ which is neither a Mersenne prime (because the distance to the nearest power of two exceeds one) nor a Sophie Germain prime (because $2m+1 = 2043$ is not prime).

Write $m = 2^{10} - 3$, i.e., $q = 10$ and $k = 3$. Notice inequality (15) is satisfied because $(10 - 1)/2 = 4 > k > 1$. Choose the multiplier $\alpha = 991$, which is the largest primitive root of 1021. Employ the seed $x_0 = 987$. Using these choices, the **MCG** (2) has period 1021. To calculate x_1 we find $k\gamma + \lambda = 3 \cdot 955 + 197 = 3062$ and in the recursive step we find the next pseudorandom number is 6 + 1014 = 1020. The largest γ computed in the first step is $987 < 2^{10} - 8$ as predicted by (13). Subtraction is needed only twice when the previous pseudorandom number is 68 and 34. In this case, subtraction is never needed when the recursive step is performed. The recursive step is needed nearly 83% of the time.

Example

Consider the prime modulus $m = 1048573$. Note $m = 2^{20} - 3$, i.e., $q = 20$ and $k = 3$. Choose the multiplier $\alpha = 2$, which is a primitive root of 1048573. Take the seed $x_0 = 1048572$. Using these choices, the **MCG** (2) has full period 1048572. Subtraction is needed only once and the recursive step is never performed! Unfortunately the sequence does not appear random since every number is twice the previous number mod m. This example demonstrates that merely having an optimal period does not provide any assurance on the quality of the sequence.

Example

Consider changing only the multiplier in the previous example, setting $\alpha = 828119$ which is a primitive root of 1048573. Using these choices, the **MCG** (2) has full period 1048572. Subtraction is needed a total of four out of 1,048,572 times, including twice after performing the recursive step (after generating the numbers 854996 and 641247). The recursive step is needed nearly 79% of the time.

Algorithm 2: MCG Computation for Primes Close to a Power of Two

---

Input: An integer x with $1 \le x \le m - 1$, a multiplier α with $1 \le \alpha < m$, positive integers q and k with $2^q - k = m$
Output: $\alpha \cdot x \bmod m$.

1. $x \leftarrow \alpha \cdot x$
2. $x \leftarrow k\gamma + \lambda$ where x has the binary representation $[\gamma \mid \lambda]$ with λ holding the q LSB of x
3. If $x > 2^q - 1$ then $x \leftarrow k\gamma' + \lambda'$ where x has the binary representation $[\gamma' \mid \lambda']$ with λ' holding the q LSB of x
4. If $x > m$ then $x \leftarrow x - m$

Efficient Implementation

When performing multiplications, overflow may result. If the computations are not performed exactly, there is no guarantee the full period predicted in theory will be achieved. Indeed, the usable sample size is typically about the square root of the possible period (or the modulus) when the modulus is large based on observations. On the CRAY XD1, the largest possible modulus is $m \approx 2^{64}$ without using more sophisticated techniques, which we reject for the sake of speed. The largest period we would normally expect is on the order of $(2^q)^{1/2}$ where the modulus $m \approx 2^q >> 2^{31}$. Realizing the full period when the modulus is large is not important. Indeed, computing 2^{64} pseudorandom numbers is not going to be done fast anytime soon! On the other hand, if the period is much smaller than $m^{1/2}$, then the sequence is much less likely to be useful regardless of the quality. Longer periods than might reasonably be expected can be realized.

Example

The CRAY **RANF** is a **MCG** given by $x_n = 44485709377909 \cdot x_{n-1} \bmod 281474976710656$, which has period 2^{46} [21]. This period seems to be valid (more on this later) using direct calculation with the usual operators (* and %) in C, i.e., perform the modulo operation in (2).

Example

Consider the prime modulus $m = 281474976597361 = 2^{48} - 113295$. Choose the multiplier $\alpha = 582167988922$ and the seed $x_0 = 281474976597360$. Then implementing the previous algorithm in C yields the period $18936324 \approx (2^{48})^{1/2}$. Performing the computations exactly (and more slowly), the period should be $281474976597360 = 2^{48} - 113296$. Using an efficient implementation based on the GMP library on the CRAY XD1, merely computing (without saving or printing) the full sequence would take about 268 days (ignoring crashes!).

Rolling A Die Fast

To meet our requirements, we declare that the computation of the next pseudorandom number is "fast" if it takes at most the same amount of time (preferably less) to call a library function such as lrand48(). We next quantitatively assess performance to demonstrate that our approach is practical. We simulate rolling a usual six-sided die $2^{29} \cdot 3$ times. We summarize the results in the following table.

LCG	Implementation in C	χ^2	Time (s)
$1327760490 \cdot x_n \bmod 2^{31}-1$	Algorithm 1	1.19	11.0
$97693434 \cdot x_n \bmod 2^{37}-25$	Algorithm 2	0.926	14.1
$27355192 \cdot x_n \bmod 2^{38}-45$	Algorithm 2	6.36	14.0
$247016489220937 \cdot x_n \bmod 2^{48}-59$	Algorithm 2	78.0	13.2
$14022294538115072 \cdot x_n \bmod 2^{55}-55$	Algorithm 2	6.74	13.2
$10337092905140992 \cdot x_n \bmod 2^{56}-5$	Algorithm 2	5.69	13.2
$9853084386742924 0 \cdot x_n \bmod 2^{57}-13$	Algorithm 2	4.95	13.2
$72103240369675328 \cdot x_n \bmod 2^{58}-27$	Algorithm 2	4.53	13.2
$220959232295413228 0 \cdot x_n \bmod 2^{61}-1$	Algorithm 1	3.91	11.0
$504813132987424512 9 \cdot x_n \bmod 2^{63}-25$	Algorithm 2	2.05	13.2
RANF: $44485709377909 \cdot x_n \bmod 2^{48}$	44485709377909*xn % m	1610612748	58.9
$25214903917 \cdot x_n + 11 \bmod 2^{48}$	lrand48()	4.35	32.4
$25214903917 \cdot x_n + 11 \bmod 2^{48}$	drand48()	2.70	50.9

The die is determined using modular arithmetic, e.g., lrand48() % 6 + 1, except in the implementation that uses drand48() for which the side is 6*drand48() + 1. The chi-square χ^2 statistic is computed assuming a uniform distribution (we expect to roll each side of the die approximately 2^{28} times.) The CRAY RANF function only rolls three of the six possible outcomes (which three sides depends on the seed)! Note we are using usual C operators (not calling the FORTRAN routine) so that we can compare the cost savings by not performing the modulo operation. Both Algorithms 1 and 2 are implemented in a straightforward manner. Except for the RAND48 library calls, we seed the generator using $m - 1$, where m is the modulus. We initialize drand48() and lrand48() by calling seed48(A), where A denotes the array of shorts [0x1234, 0xabcd, 0x330e].

A fast implementation of Algorithm 1 or 2 is easily coded. The number γ is obtained by shifting q bits to the right. The integer λ is calculated by performing a bit-wise and operation with $2^q - 1$. When $q = 64$, do not apply the algorithm since γ is zero and instead use direct calculation (2). As the modulo operation is

relatively slow as we have shown, we recommend using moduli $< 2^{63}$.

Example

Consider the prime modulus $m = 18446744073709549363 = 2^{64} - 2253$. Choose the multiplier $\alpha = 1262014585074097263$ and the seed $x_0 = m - 1$. Then implementing Algorithm 2 in C produces $x_n = 0$ for all $n \geq 63$. If we abandon Algorithm 2 and perform the computations in C directly (as $x_n = \alpha * x_{n-1} \% m$), the period is long.

We see that performing the modulo operation is relatively slow. Our algorithms are fast compared to calling library functions. Using Mersenne primes (Algorithm 1), we see the speedup is about 1.2× compared to an implementation of Algorithm 2. We have settled important implementation issues to achieve a fast implementation of an **MCG**. We next address how to find constants to use in an **MCG** so that the generated sequence is high-quality.

Choosing Prime Moduli

Before we are able to select multipliers, we need to decide the value of the modulus m. We impose the requirement that $m - 1$ is easy to factor so that finding primitive roots is fast using (4). In doing so, we eliminate few, if any, moduli. It does not seem prudent to invest a great deal of effort on one modulus without any evidence it is a particularly good choice especially when it is not practical to explore all possibilities. Indeed, performing an exhaustive search in which we find the best multipliers for every valid modulus is practical only for small q. We suspect that it is harder to find especially good multipliers for some large moduli, just as not all integers are equally easy to factor.

> **Question:** Is it easier to find more multipliers that yield high-quality sequences by choosing the prime modulus m on the basis of the number of factors of $m - 1$?

The main requirement is that the modulus $m = 2^q - k$, is a prime close to a power of two satisfying inequality (15). Consider a few different strategies to select such prime modulus:

1. The prime closest (but less than) a power of two.
2. A special prime such as a Mersenne or Sophie Germain prime closest (but less than) a power of two.
3. The prime furthest and close enough to a power of two.
4. A prime m for which $m - 1$ has the least factors.
5. A prime m for which $m - 1$ has the most factors.

The first few choices are rather arbitrary choices which are quickly discovered. Although some authors advocate choosing a prime closest to a power of two because the period is longer in theory, this notion is flawed because in practice the period is usually much smaller for large q and may even be longer in practice for smaller moduli. We hope an **MCG** is capable of producing more pseudorandom numbers than will be used by an application. So it should not matter if the period is negligibly smaller. We stress that merely increasing the size of the moduli does not provide any assurance the period will be longer in practice.

Recall, it has been argued that using Sophie Germain primes as prime moduli makes it easy to find primitive roots [9]. However, it is not the prime modulus m which should be a Sophie Germain prime, but rather $(m - 1)/2$ which should be a Sophie Germain prime. Regardless, it is equally easy to find primitive roots for a far greater number of primes, e.g., there are many more primes than Sophie Germain primes of the form $2^a p^b + 1$, where p is prime, and all such primes offer the same advantages. Moreover, finding small primitive roots is fast for any choice of modulus unless it is difficult to factor $m - 1$ (which we have already addressed). Furthermore, there is little need for optimization for such task which is performed only once for any choice of modulus. Since special primes are included in the other strategies (1,4), we shall focus on them.

The antithesis of using Sophie Germain primes is listed last (item 5). Our reasoning for using such moduli is based on the large set (9). For large q, there are a large number of exponents to check. Is it possible to more quickly find good multipliers by eliminating more of the exponents?

To define an algorithm that finds moduli, it is necessary to specify how to break ties when more than one moduli might meet the stated criteria. When using the number of factors as a criteria (items 4-5) and more than one modulus has the optimal number of factors, we arbitrarily choose either the smallest moduli whose largest factor is minimal, or the largest moduli (which is well-defined because extrema always exists on a finite set). We say an integer "factors quickly" if we can can completely factor in some predetermined fixed number of steps using some particular program. The following algorithm is easily adapted for all of the described strategies (and others as well):

Algorithm 3: Find Prime Modulus *m* Based on Factors of *m* − 1

Input: A positive integer q

Output: The smallest modulus m having the most factors with the smallest largest factor, $|2^q - m| < 2^{(q-1)/2}$

1. $k \leftarrow 2^{(q-1)/2} - 1$
2. $n \leftarrow 2^q - 1 - k$
3. $n \leftarrow$ next prime greater than n
4. $max_{factors} \leftarrow 0$
5. While $n < 2^q$ carry out steps a-b:
   a. if $n - 1$ factors quickly then
      if the number of factors exceeds $max_{factors}$ then
         $m \leftarrow n$
         $max_{factors} \leftarrow$ number of factors
         $largest_{factor} \leftarrow$ largest value among the factors
      else if the number of factors equals $max_{factors}$ then
         if the largest factor is less than $largest_{factor}$ then
            $m \leftarrow n$
            $largest_{factor} \leftarrow$ largest value among the factors
   b. $n \leftarrow$ next prime greater than n

Note that after step 2, n is even (since both $2^q - 1$ and k are odd) and so we are certain the next prime is larger than n and within the required distance. We have implemented Algorithm 3 in C using the GMP library and our own version of SQUFOF. We list some results in tables below. It is interesting that the primes seem to occur in a certain "regularity" in the sense for example the distance k generally increases with q when searching for the smallest prime m sufficiently near 2^q for which $m - 1$ has the least or most factors.

Prime Moduli $m = 2^q - k$

q	Small moduli	Moduli m for which $m - 1$ has the most factors with small largest factor	Large moduli m for which $m - 1$ has the most factors
	k	k	k
31	32725	25477	21757
32	32759	15095	135
33	65529	15781	15781
34	65513	11733	3753
35	131055	4189	4189
36	131057	20895	1995
37	262143	63883	6133
38	262125	51075	51075
39	524281	7	7
40	524255	292125	34595
41	1048539	771529	31
42	1048571	834771	51933
43	2097121	404167	404167
44	2097137	1242069	30705
45	4194283	2484139	674821
46	4194285	4143213	4143213
47	8388535	2134177	564427
48	8388575	6805845	1128855
49	16777171	15282151	15282151
50	16777133[1]	13683813	13683813
51	33554409	22153957	22153957
52	33554399	28334685	10193835
53	67108861	49350691	28197421
54	67108773	23207793	23207793
55	134217675	66709861	1597957
56	134217723	92831175	92831175
57	268435401	98575351	98575351
58	268435415[1]	197150703	197150703
59	536870907	200307607	102293257
60	536870903	84750435	84750435
61	1073741719	968978011	968978011
62	1073741781	947209497	3723723
63	2147483637	1894418995	10258255
64	2147483609	1083182115	673280805

[1] Sophie Germain prime

Prime Moduli $m = 2^q - k$

q	Moduli m for which $m-1$ has two factors with small largest factor k	Large m for which $m-1$ has two factors k	Large moduli k
31	10239[1]	69	1[2]
32	12287	209	5[1]
33	34815	9	9
34	43007	641	41
35	87039	519	31
36	12287	137	5
37	262143	45	25
38	110591	401	45
39	471039	135	7
40	190463	437	87
41	864255	75	21
42	270335	2201[1]	11
43	1318911	291	57
44	552959	1493	17
45	1146879[1]	573	55
46	3244031	857	21
47	5373951	771	115
48	4890623	1823	59
49	4980735	2295	81[1]
50	15679487	161	27
51	6553599	465	129
52	24575999	473	47
53	51380223	1269	111
54	19464191	1031[1]	33
55	81657855	579	55
56	105381887	2249	5
57	215351295	423	13
58	242221055	137	27
59	268435455	99	55
60	364904447	107	93
61	570425343	2373	1[2]
62	987758591	791	57
63	117440511	915	25
64	1676673023	1469	59

[1]Sophie Germain prime [2]Mersenne prime

“Small” and “large” are used in this context instead of smallest and largest, respectively, because the SQUFOF algorithm is not always guaranteed to factor completely, even though we have conducted an exhaustive search. Nevertheless, a quick inspection of the values of k indicates the observed values are in the range of the expected values. Moreover, the number of factors is always consistent with observed values. In particular, note that 2 is always a factor of $m - 1$ for all prime moduli m. In any case, we have

explicitly provided a list of many moduli (among the large number of valid moduli) suitable for further investigation.

Primitive Roots

After having chosen a prime modulus, the task is to find a primitive root α. Based on our experiments and known theoretical results, the smallest primitive root works as well as any other root when using (9) to find good multipliers, which is the topic of the next section. It is easy to find such roots [22]. Based on inequality (4), we next present an algorithm which is easily adapted to find various primitive roots.

Algorithm 4: Find Smallest Primitive Root

Input: A prime modulus m for which $m - 1$ factors easily
Output: A small primitive root α of m

1. Compute an array of distinct factors F[1..n] such that $k \mid (m - 1)$ if and only if k equals F[i] for some i with $1 \le i \le n$.
2. Set $e_i = (m - 1)/F[i]$ for $1 \le i \le n$.
3. Put Prime ← 2
4. Let $i \leftarrow 1$
5. While $i \le n$ carry out steps a-b:
   a. $r \leftarrow \text{Prime}^{e_i} \bmod m$
   b. if r equals 1 then
      Set Prime ← the next prime greater than Prime.
      $i \leftarrow 1$
6. α ← Prime

Typically the primitive root is quite small so that a small array may be used to hold small primes (or call a function in a library such as GMP to obtain the next prime). There is a subtle point to notice about this algorithm. It is not possible to employ usual library functions to raise primes to large powers (modulo m). At least one of the exponents is large because 2 divides $m - 1$ for large primes m. We developed our own code and found by comparison that the GMP library provides a suitably efficient exponentiation function. We list the primitive roots for the moduli in the preceding tables below. For theses moduli it is striking that there are always primitive roots less than or equal to seven when there is only one other factor besides 2, and that 3 is always a primitive root when the only other factor besides 2 is relatively small.

Primitive Roots α of Prime Moduli *m*

	Small moduli		Moduli *m* for which *m* − 1 has the most factors with small largest factor		Large moduli *m* for which *m* − 1 has the most factors	
q	*m*	*α*	*m*	*α*	*m*	*α*
31	2147450923	3	2147458171	3	2147461891	3
32	4294934537	3	4294952201	3	4294967161	67
33	8589869063	5	8589918811	19	8589918811	19
34	17179803671	7	17179857451	2	17179865431	11
35	34359607313	3	34359734179	2	34359734179	2
36	68719345679	17	68719455841	23	68719474741	2
37	137438691329	3	137438889589	2	137438947339	2
38	274877644819	2	274877855869	13	274877855869	13
39	549755289607	3	549755813881	11	549755813881	11
40	1099511103521	3	1099511335651	3	1099511593181	2
41	2199022207013	2	2199022484023	3	2199023255521	29
42	4398045462533	2	4398045676333	5	4398046459171	2
43	8796090925087	5	8796092618041	83	8796092618041	83
44	17592183947279	7	17592184802347	3	17592186013711	67
45	35184367894549	17	35184369604693	2	35184371414011	2
46	70368739983379	2	70368740034451	2	70368740034451	2
47	140737479966793	5	140737486221151	7	140737487790901	2
48	281474968322081	11	281474969904811	2	281474975581801	83
49	562949936644141	13	562949938139161	19	562949938139161	19
50	1125899890065491[1]	2	1125899893158811	7	1125899893158811	7
51	2251799780130839	7	2251799791531291	2	2251799791531291	2
52	4503599593816097	3	4503599599035811	7	4503599617176661	23
53	9007199187632131	2	9007199205390301	19	9007199226543571	2
54	18014398442373211	2	18014398486274191	11	18014398486274191	11
55	36028796884746293	2	36028796952254107	3	36028797017366011	11
56	72057593903710213	2	72057593945096761	47	72057593945096761	47
57	144115187807420471	13	144115187977280521	31	144115187977280521	31
58	288230375883276329[1]	3	288230375954561041	23	288230375954561041	23
59	576460751766552581	2	576460752103115881	29	576460752201130231	19
60	1152921504069976073	3	1152921504522096541	13	1152921504522096541	13
61	2305843008139952233	5	2305843008244715941	31	2305843008244715941	31
62	4611686017353646123	2	4611686017480178407	7	4611686018423664181	73
63	9223372034707292171	13	9223372034960356813	2	9223372036844517553	5
64	18446744071562068007	5	18446744072626369501	29	18446744073036270811	19

[1] Sophie Germain prime

Primitive Roots of Prime Moduli *m*

	Moduli *m* for which *m* − 1 has two factors with small largest factor		Large *m* for which *m* − 1 has two factors		Large moduli	
q	*m*	*α*	*m*	*α*	*m*	*α*
31	2147473409[1]	3	2147483579	2	2147483647[2]	7
32	4294955009	3	4294967087	5	4294967291[1]	2
33	8589899777	3	8589934583	5	8589934583	5
34	17179826177	3	17179868543	5	17179869143	5
35	34359651329	3	34359737849	3	34359738337	5
36	68719464449	3	68719476599	7	68719476731	2
37	137438691329	3	137438953427	2	137438953447	5
38	274877796353	3	274877906543	5	274877906899	2
39	549755342849	3	549755813753	3	549755813881	11
40	1099511437313	3	1099511627339	2	1099511627689	13
41	2199022391297	3	2199023255477	2	2199023255531	2
42	4398046240769	3	4398046508903[1]	5	4398046511093	2
43	8796091703297	3	8796093021917	2	8796093022151	7
44	17592185491457	3	17592186042923	2	17592186044399	7
45	35184370941953[1]	3	35184372088259	2	35184372088777	13
46	70368740933633	3	70368744176807	5	70368744177643	2
47	140737482981377	3	140737488354557	2	140737488355213	5
48	281474971820033	3	281474976708833	3	281474976710597	2
49	562949948440577	3	562949953419017	3	562949953421231[1]	17
50	1125899891163137	3	1125899906842463	5	1125899906842597	41
51	2251799807131649	3	2251799813684783	5	2251799813685119	11
52	4503599602794497	3	4503599627370023	5	4503599627370449	3
53	9007199203360769	3	9007199254739723	2	9007199254740881	3
54	18014398490017793	3	18014398509480953[1]	3	18014398509481951	3
55	36028796937306113	3	36028797018963389	2	36028797018963913	7
56	72057593932546049	3	72057594037925687	5	72057594037927931	11
57	144115187860504577	3	144115188075855449	3	144115188075855859	2
58	288230375909490689	3	288230376151711607	5	288230376151711717	17
59	576460752034988033	3	576460752303423389	2	576460752303423433	5
60	1152921504241942529	3	1152921504606846869	2	1152921504606846883	2
61	2305843009069514753[1]	3	2305843009213691579	2	2305843009213693951[2]	37
62	4611686018080571393	3	4611686018427387113	3	4611686018427387847	17
63	9223372036737335297	3	9223372036854774893	2	9223372036854775783	3
64	18446744072032878593	3	18446744073709550147	2	18446744073709551557	2

[1]Sophie Germain prime [2]Mersenne prime.

Finding Multipliers Using LLL Reduction

After having determined a prime modulus and corresponding primitive root, the task is to select a set of multipliers $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ suitable for pseudorandom number generation and especially for parallel processing. The trick is to select those powers of the primitive root α (i.e., $\alpha_k = \alpha^r$ for some integer *r*) which yield high quality sequences [9][10]. In the sequel, we investigate this approach.

Precisely what is a high quality sequence depends on the application. A valid approach is to assess the

quality based on observations of the results of experimental runs (i.e., based on an application basis using empirical testing). Notwithstanding this caveat, it is widely accepted that good constants for an **MCG** will yield good spectral results, which is a measure of the smallest distance between so-called hyperplanes (determined by overlapping tuples) with smaller distances being better (because the gaps are smaller) [1]. Typically the number of dimensions (length of tuple sequence) is between six and eight. The shortest distance is usually %ldquo;normalized" between 0 and 1. Spectral results have been used for example to analyze results for parallel computations using the CRAY **RANF** pseudorandom number generator on a CRAY system [23].

It is widely accepted that computing spectral results is time-consuming. It seems costly to check the quality of a large number of primitive roots based on spectral results for large moduli. To reduce the time to assess the quality of a multiplier, an approximation is calculated using the NTL library to perform an LLL reduction (named after the discovers Lenstra, Lenstra and Lovász) [24] [25] [26] [27] [28]. We next briefly review the calculation of the approximation.

As usual, let m denote a prime modulus and α a candidate multiplier. We consider only eight dimensions since the approximation is fast compared to computing the exact result, which is typically based on lower dimensions [27]. Choose a dimension $d \in \{2,3,4,6,7,8\}$. Form a $d \times d$ matrix U_d by taking the first d rows and columns of the following matrix:

$$U_8 = \begin{bmatrix} m & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\alpha & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\alpha^2 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -\alpha^3 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ -\alpha^4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ -\alpha^5 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ -\alpha^6 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ -\alpha^7 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Note a suitable exponentiation function is needed to raise a large multiplier to small powers (or perform multiplication exactly). The NTL library provides such functions plus a routine to perform LLL reduction, which performs elementary row transformations so that the rows form an LLL reduced basis (see LLL.txt in the NTL documentation). After performing LLL reduction on the matrix U_d, the minimum of the inner product of each row with itself is multiplied by a precomputed constant, which depends on the modulus and the chosen dimension d to obtain a result for the chosen dimension. The minimum of all such results over all dimensions yields the final approximation. Larger results closer to one are better. Values over 0.74 are less commonly observed and values around 0.77 or higher are especially rare.

We search for multipliers that have relatively high normalized results. We propose to limit the search by setting a slightly higher threshold on the minimum for the first few dimensions (2 through 6). In other words, a lower bound on the minimum for the last couple dimensions (7 and 8) is not set quite as high as for the lower dimensions. In this way, we hope to find a larger number of multipliers by eliminating fewer multipliers for which the approximations are relatively high based only on the lower dimensions. We next present our algorithm to find many multipliers, which is similar to an implementation that is designed to find a single multiplier by raising the primitive root to suitable powers until good LLL-spectral results are realized (see lll_search.c) [28].

Algorithm 5: Select Primitive Roots

Input: A prime modulus m and a primitive root α of m
Output: A set of multipliers and corresponding normalized LLL-spectral results

1. Calculate "normalization factors:"

$N_2 = 0.75^{1/2}/m$, $N_3 = (0.5^{1/2}/m)^{2/3}$, $N_4 = (0.5/m)^{1/2}$, $N_5 = (0.125^{1/2}/m)^{2/5}$,
$N_6 = (0.046875^{1/2}/m)^{1/3}$, $N_7 = (0.125/m)^{2/7}$, $N_8 = (0.0625/m)^{1/4}$;

Choose limits:
$Max_{approximations} \leftarrow 536870912$,
$Max_{roots} \leftarrow 4096$,
$Min_{2-6} \leftarrow 0.5476$,
$Min_{7-8} \leftarrow 0.4489$;

Set counters:
$e = 0$,
$i \leftarrow 0$,
$c \leftarrow 0$.

2. While $c < Max_{approximations}$ carry out steps a-f:
   a. $c \leftarrow c + 1$
   b. repeat
      $e \leftarrow e + 1$
      if e equals $m - 1$, proceed to next step (3)
      until gcd($m - 1$, e) is not equal to one
   c. root $\leftarrow \alpha^e$
   d. Set $Min \leftarrow 2$,
      $Minimum \leftarrow Min_{2-6}$,
      $d \leftarrow 2$.
   e. while Min ≥ Minimum and $d \leq 8$
      $V \leftarrow U_d$ transformed into an LLL-reduced basis
      $\delta_d \leftarrow \min \{ v_0 \cdot v_0, v_1 \cdot v_1, \ldots, v_{d-1} \cdot v_{d-1} \}$, where v_r is the r^{th} row of V
      $\eta \leftarrow N_d \cdot \delta_d$
      if $\eta <$ Min then Min $\leftarrow \eta$
      if d equals 7 then $Minimum \leftarrow Min_{7-8}$
      $d \leftarrow d + 1$
   f. if Min ≥ Minimum then
      $Root_i$ = root
      $Result_i = Min^{1/2}$
      $i \leftarrow i + 1$
      if i equals Max_{roots}, proceed to next step (3)
3. If $i > 0$, merge sort Root using Result to order elements

---

We employ a lazy evaluation technique by aborting (step e) as soon as a small value is discovered instead of waiting until the minimum is computed. We also compute the entries of the matrix U_8 only once and reuse the entries. The normalization factors may be calculated using standard library functions (using powf() in C instead of using higher precision as in type RR in NTL library). Since these factors are computed only once, performance is not relevant. We have squared the constants to avoid computing square roots, which improves performance slightly, e.g., $0.5476 = (0.74)^2$ and $0.4489 = (0.67)^2$. We compute square roots mainly for convenience (step f). We use the LLL_FP routine instead of LLL routine, which significantly improves performance. We also used the GMP library to perform initial computations (instead of using NTL library functions). We found 4096 multipliers for the Mersenne modulus $2^{61} - 1$ in about 6.67 hours instead of 11.8 hours by implementing these optimizations.

Increasing the given limits on Min_{2-6} or Min_{7-8} will significantly reduce the number of multipliers found. Increase Max_{roots} (for large moduli) to find more multipliers with higher LLL-spectral results and ignore those multipliers with smaller LLL-spectral results, which is the reason for sorting. There are only finitely many such results and for this reason we include other counters. Reduce $Max_{approximations}$ to run the algorithm faster, which yields fewer results. We provide a code fragment in C below.

Code Fragment in C Implementing Algorithm 5 to Find Many Multipliers

```
unsigned long int r = 0;
mpz_t gmp_root, gmp_base, gmp_mod, gmp_n1, gmp_r[9] ; mpz_init( gmp_root );
mpz_init( gmp_base ); mpz_set_ui( gmp_base, PrimitiveRoot );
mpz_init( gmp_mod ); mpz_set_ui( gmp_mod, modulus );
mpz_init( gmp_n1 ); mpz_set_ui( gmp_n1, N1 ); // N1 = modulus - 1
for (int j = 1; j < 9; i++ ) mpz_init( gmp_r[j] );
unsigned long int Root;
ZZ multiplier, m;
conv(m,modulus );
float N[9];
dm = (float)modulus;
N[2] = powf( 0.75, 0.5) / dm;
N[3] = powf( pow(0.5,0.5) / dm , 0.6666667 );
N[4] = powf( 0.5 / dm , 0.5 );
N[5] = powf( powf(0.125,0.5) / dm , 0.4 );
N[6] = powf( powf(0.046875,0.5) / dm , 0.3333333 );
N[7] = powf( 0.125 / dm , 0.28571429 );
N[8] = powf( 0.0625 / dm , 0.25 );
ZZ products[9];
const float Min26 = 0.5476, // square of lower acceptable limit for dim 2-6
      Min78 = 0.4489; // square of acceptable limit for dim 7-8
const float Minimum[10] = {0,0,Min26,Min26,Min26,Min26,Min26,Min26,Min78,Min78};
int i = 0, CounterLLL;
for ( CounterLLL = 0; CounterLLL < Max_approximations ; CounterLLL++ ) {
      for( r++; mpz_gcd_ui( NULL, gmp_n1, r ) != 1UL; r++ ); // gcd
      if ( r >= N1 ) break;
      Root = mpz_get_ui ( gmp_root );
      multiplier = to_ZZ( Root );
      float norm, min = 2;
      unsigned long int dprev = 1, d = 2;
      for ( ; ( min >= Minimum[d] ) && (d <= dim); dprev = d, d++) {
            products[dprev] = products[dprev-1] * multiplier;
            mat_ZZ x;
            x.SetDims(d,d);
            x[0][0] = m;
            for (int j = 1; j < d; j++) {
                  x[j][j] = 1;
                  x[j][0] = products[j];
            }
            LLL_FP( x, 0.75, 0, NULL, 0);
            float min_d = to_float( x[0] * x[0] );
            for (int j = 1; j < d; j++) {
                  float ip = to_float( x[j] * x[j] );
                  if (min_d > ip) min_d = ip;
            }
            norm = N[d] * min_d;
            if (norm < min) min = norm;
```

```
            }
            if( min >= Minimum[d] ) {
                  Root[ i ] = Root;
                  Result[ i ] = sqrt(min);
                  i++; if (i == Max_roots) break;
            }
      }
      if( i ) { // sort and print }
```

Using the preceding algorithm, we found at least 4096 multipliers for all moduli near $m \approx 2^q$ with $31 \le q \le 64$. In particular, we found all except one of the multipliers for moduli near $m \approx 2^q$ with $31 \le q \le 36$ listed in [27], which indicates our implementation is correct. Moreover, we found multipliers with higher LLL-spectral results for all q. Based on the LLL-spectral results alone, we cannot distinguish one class of moduli as particularly a good choice. For example, if we count the number of multipliers with an LLL-spectral result above a certain value, then the class that produces the largest number of multipliers exceeding the prescribed value varies depending on both the given value and the power of two for which the moduli are near. Hence, it seems LLL reduction is not useful to choose prime moduli m on the basis of the number of factors of $m - 1$, which is consistent with Leonard Euler's statement that [29]

> "Mathematicians have tried in vain to this day to discover some order in the sequence of prime numbers, and we have reason to believe that it is a mystery into which the human mind will never penetrate."

Nevertheless, some other means such as empirical testing may be useful to select prime moduli and for any choice of prime moduli, LLL reduction is useful to find candidate multipliers.

Selecting Multipliers Using Empirical Tests

Given a list of multipliers, we cannot simply select those multipliers with high LLL-spectral results. There are two potential problems. One problem is the period may be unacceptably short, which may be a consequence of the method of computation. Another problem is the sequence may not appear sufficiently random.

Checking the period of a large list of multipliers is not a trivial task. Which cycle finding algorithm is fastest? This question does not seem to be settled in the literature. It is claimed that the “multi-stack” algorithm is about 20% faster on average than Brent's algorithm [30] [31]. We confirm that if the period is short, it can be discovered fairly quickly using the multi-stack algorithm. However, we also found many cases for which it is considerably faster to calculate the period using Brent's algorithm [32].

Example

Modulus	Multiplier	Period	Time (s)	
			multi-stack	Brent
$2^{33} - 9$	26891986	8589934582	260	104
$2^{33} - 9$	8137022074	19739	0.00230	0.00125
$2^{31} - 1$	1977654935	2147483646	61.9	32.6

Another advantage of Brent's algorithm is that it has minimal memory requirements. Intuitively it is plain the time required to detect a cycle is directly proportional to the period (since it seems it is necessary to calculate enough elements before encountering a repeated value). Hence, it is not practical to calculate the period whenever it is close to 2^q for $q >> 33$. For example, a multi-stack program may not halt for over a week to find the period for a single multiplier; whence, it would take impossibly long to calculate the period for thousands of multipliers even when many of them provide a short period. We discovered a

simple modification of Brent's algorithm that not only finds the correct period when it is not too long based on extensive testing, but also terminates in a reasonable amount of time even when the period is too long to be calculated exactly. Furthermore, this modification has negligible effect on the running time. We present Brent's modified algorithm next.

Algorithm 6: Find Short Period of an LCG

Input: The function *f()* that computes the next pseudorandom number and the seed *x*
Output: *k*, which is either the period or a larger integer indicating a shorter period was not found

1. Put $r \leftarrow 1$
2. Set $k \leftarrow 1$
3. Put $y \leftarrow f()$
4. While $x \neq y$ carry out steps a-c:
   a. if r equals k then
      Set $x \leftarrow y$
      Set $r \leftarrow 2r$
      if r equals 2^{34} then proceed to next step (5)
      $k \leftarrow 0$
   b. $y \leftarrow f()$
   c. $k = k + 1$
5. If r equals 2^{34} then $k \leftarrow$ the largest integer possible

---

Example

We used Algorithm 6 to verify that we could not find a short period $<< 2^{46}$ for the CRAY **RANF** **MCG** in approximately 10 minutes 20 seconds [21].

Note the period is often dependent of the seed. Since a long period does not ensure the generated sequence meets the requirements of an application, we propose running empirical tests to check the quality of the sequences. The proposed strategy is to conduct multiple tests based on reasonably large subsequences. If the period is extremely short, then the tests are more likely to detect poor characteristics of the generated sequences.

There exist well-known tests such as the “diehard” tests which yield statistical data based on sound theory [33]. While it is practical to carry-out such tests for a small number of multipliers, it is time-consuming to inspect a large amount of data for a large number of multipliers. We seek a faster method to empirically test multipliers found using LLL reduction.

We next describe our method to filter out good multipliers. We conducted the following common tests by successively generating subsequences of length 6881280 ten times without reseeding for each multiplier found:

Type of Test	Number of Tests
Permutation	Four using *k*-tuples, *k* = 5,6,7,8
Uniformity	One
Independence	Six using *k*-dimensional hypercubes, *k* = 2,3,4,5,6,7

The permutation test records the order (which should be uniformly distributed) among the possible *k!* orderings of *k*-tuples. The uniformity test counts how many observations fall into a fixed number of equal-sized bins. The *k*-dimensional independence test records the number of hits in *k*-dimensional equal-sized hypercubes. We reuse the generated data for all tests (i.e., we use the same data for the 5-tuple permutation test as for the uniformity test, etc.) without overlapping (independent samples), e.g., the first two *k*-tuples involve the first $2 \cdot k$ numbers generated.

For each of the 110 tests, we assign a statistic σ using a χ^2 goodness-of-fit test assuming the sample comes from a uniform distribution as follows:

σ	Region of χ^2 statistic
0	$10\% \le \chi^2 \le 90\%$
1	$5\% \le \chi^2 < 10\%$ or $90\% < \chi^2 \le 95\%$
2	$1\% \le \chi^2 < 5\%$ or $95\% < \chi^2 \le 99\%$
3	$\chi^2 < 1\%$ or $\chi^2 > 99\%$

We devised this statistic based on Knuth's discussion of “general test procedures” [1]. We compute the χ^2 statistic based on the percentage points of the χ^2 distribution using MQLS software [34]. The final statistic is the sum $\varsigma = \Sigma\, \sigma$ where the summation is taken over the 110 tests. The statistic will depend on the seed employed. In our tests, we set the seed to $m - 1$, where m is the prime modulus. We accept those multipliers which have relatively low sums. We propose to reject whenever we find “outliers” more than 20% of the time, where the statistic $\sigma > 0$ for any outlier. The average statistic σ for an outlier is 2 and so the sum of the statistic σ for any particular test over the ten trials should not exceed $(.20)10 \cdot 2 = 4$. In our case, we run eleven tests and so we propose to reject whenever the final statistic $\varsigma > 4 \cdot 11 = 44$.

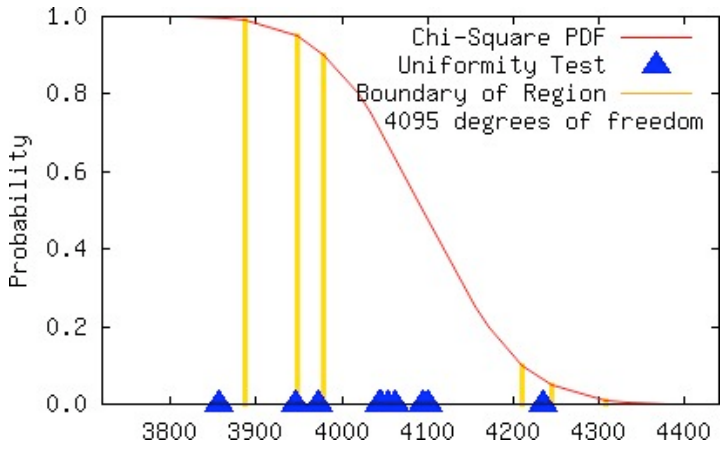
What is the justification for this testing procedure? We normally expect to see the χ^2 statistic to appear uniformly randomly distributed when the generated data appears uniformly randomly distributed. This means we also expect to see outliers occasionally. Ideally we would graph the results instead of computing the sum.

Example

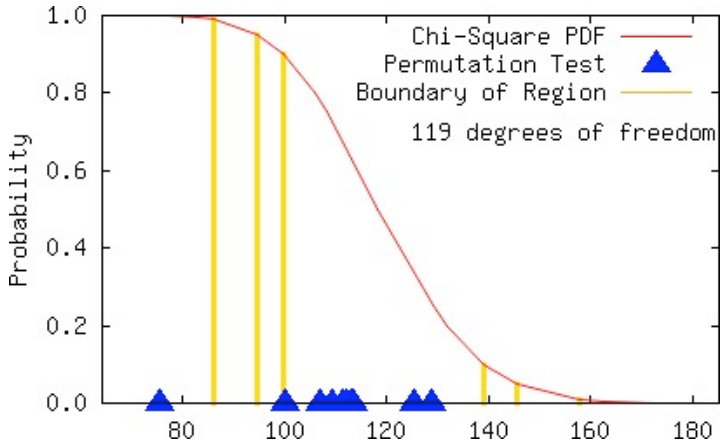
Several multipliers were found using LLL reduction for the prime moduli $2^{31} - 1$ and $2^{33} - 9$ [27]. We list our results for all of these multipliers and a couple other multipliers which we found below for comparison purposes.

Modulus	Multiplier	Period	LLL-spectral Result	ς
$2^{31} - 1$	598753959[27]	2147483646	0.734350	43
$2^{31} - 1$	117879879[27]	2147483646	0.743094	36
$2^{31} - 1$	629824009[27]	2147483646	0.748798	47
$2^{31} - 1$	1355089539[27]	2147483646	0.749724	34
$2^{31} - 1$	1101592370	2147483646	0.761410	17
$2^{33} - 9$	8137022074[27]	19739	0.753160	330
$2^{33} - 9$	26891986	8589934582	0.756007	40

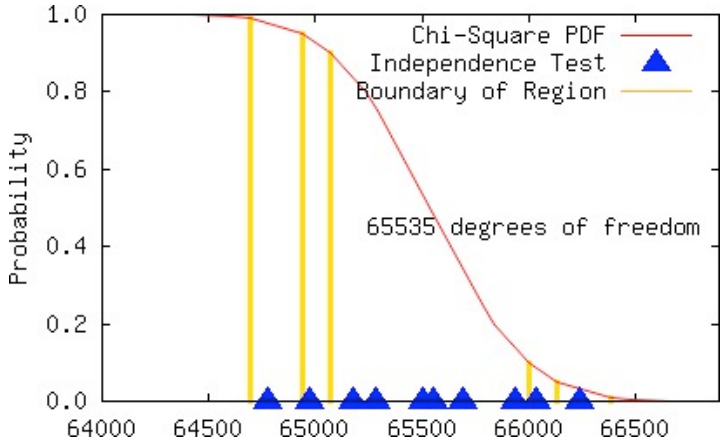
The multiplier 26891986 for the modulus $2^{33} - 9$ yields not only a full period but also a relatively small sum ς (40 << 330). We provide a few graphs below to visually show that the individual statistics are typically scattered when the multiplier is a good choice.



Chi-square statistics for prime modulus $2^{33}-9$ and multiplier 26891986 in uniformity test

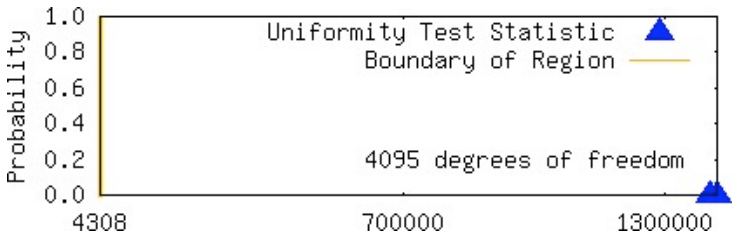


Chi-square statistics for prime modulus $2^{33}-9$ and multiplier 26891986 in 5-tuple permutation test

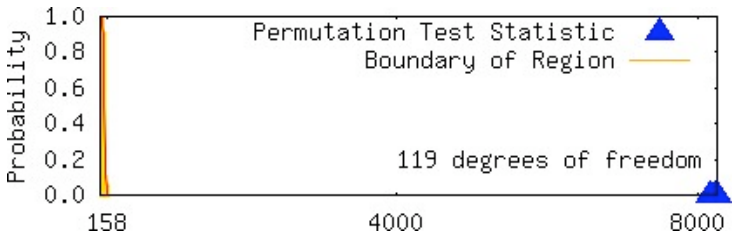


Chi-square statistics for prime modulus $2^{33}-9$ and multiplier 26891986 in 2D independence test

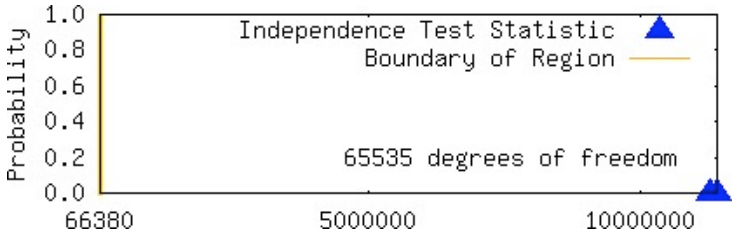
In stark contrast, the multiplier 8137022074 for the modulus $2^{33} - 9$ not only yields not only an unacceptably short period but also a relatively large sum ς (330 >> 40). We see that the results are not scattered in the proper regions.



All statistics for prime modulus $2^{33}-9$ and multiplier 8137022074 are outliers in uniformity test



All statistics for prime modulus $2^{33}-9$ and multiplier 8137022074 are outliers in 5-tuple permutation test



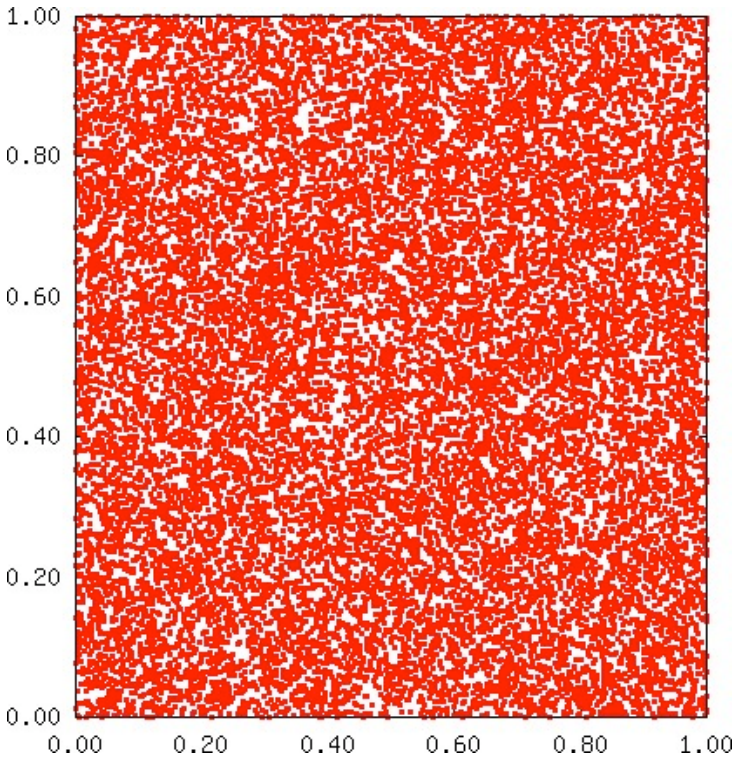
All statistics for prime modulus $2^{33}-9$ and multiplier 8137022074 are outliers in 2D independence test

We find that the **MCG** having the multiplier 26891986 and the modulus $2^{33} - 9$ clearly passes nearly all of the diehard tests. Using the multiplier 8137022074, the **MCG** clearly fails most of the diehard tests. Based on these results, the multiplier 8137022074 is a poor choice for the modulus $2^{33} - 9$ even though the LLL-spectral result is relatively high.

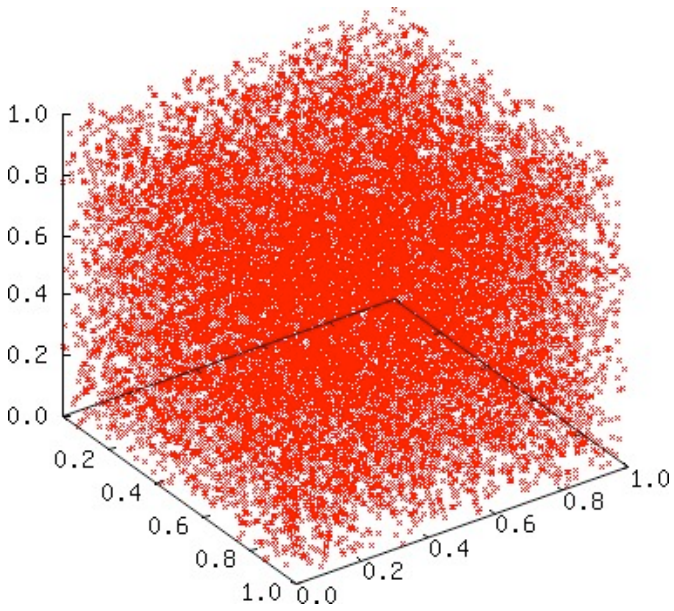
It is generally observed that poorly chosen constants for an **LCG** can be detected by a visual inspection of 2D or 3D graphs. Although the multiplier 8137022074 yields a 2D plot that does exhibit subtle patterns that might perhaps lead us to reject the multiplier, the 2D and 3D plots in this instance do not exhibit large gaps, which normally would be anticipated whenever the constants for the **MCG** are poorly chosen, i.e., the plots are unexpectedly good (because the empirical results are unsatisfactory), although such graphs are consistent with high LLL-spectral results. Below we plot 19739 points using the sequence

$$s = \{ \frac{1}{m-1}(x_0, x_1, x_2, \ldots, x_{19738}) \},$$

where m is the modulus, $x_0 = m - 1$ is the seed, and x_i is the i^{th} pseudorandom number generated by the **MCG**.



2D plot of $\{(s_n, s_{n+1}) \mid 0 \le n \le 19738\}$ for the rejected multiplier 8137022074



3D plot of $\{(s_n, s_{n+1}, s_{n+2}) \mid 0 \le n \le 19737\}$ for the rejected multiplier 8137022074

Surprisingly, for the vast majority of the multipliers found using LLL reduction for the modulus $2^{33} - 9$, the statistic ς is relatively large. In other words, there are far fewer multipliers with a relatively small statistic in this case. We observe this phenomenon only for intermediate moduli (our tests are limited). For instance, for the prime modulus $2^{38} - 45$, only two of the 4096 multipliers found using LLL reduction may be considered satisfactory using our testing procedure! When the modulus is smaller, as in the case $m = 2^{31} - 1$, or larger than 2^{49}, we find far fewer large values of the statistic ς.

Interestingly, we found relatively more satisfactory multipliers (about 900 in each case) for moduli close to

$m = 2^{33}$ for which $m - 1$ either has many factors or a relatively small factor besides two. Although we found over 8000 multipliers for both moduli $m = 2^{39} - 7$ and $m = 2^{39} - 524281$ with high LLL-spectral results, few passed our empirical tests and those that did pass our tests failed to provide a full period. For example, $m = 2^{39} - 7$, the multiplier 407569451297 has an LLL-spectral result of 0.736035 and a low ς = 4, which are satisfactory but the period is only 7151242. For the modulus $m = 2^{39} - 524281$, the multiplier 107627735285 has an LLL-spectral result of 0.676041 and a ς = 35 and the period is 13158220, which is relatively long (not a full period) compared to other multipliers that pass out tests. Thus the problem is not limited by the small quantity of satisfactory empirical results. Often the period is too short even when the LLL-spectral and empirical results are satisfactory. In the case of $m = 2^{49} - 81$, we find many multipliers for which the LLL-spectral and empirical results are acceptable but for which the period is short (around 50+ million or less, which may be acceptable for some applications).

In review, we search for multipliers using our algorithm based on a fast implementation of LLL reduction. We then filter the multipliers by eliminating the ones that have either unacceptable empirical results based on a series of tests or too short period based on our modified version of Brent's algorithm. We provide a partial list of multipliers that pass our testing procedure for a few moduli in the table below. While these multipliers are good choices, our tests are not exhaustive and we could find better choices.

Multipliers for Prime Modulus $m = 2^{31} - 1$

1327760490	2066353702	64673635	13496534	2115602258
1130487585	366987602	449666487	1179466207	1610952690
1144912594	750812715	237482926	589590877	63009789
1646375777	1982352019	276600333	1340326634	1172820338
849788117	1342204164	109002162	658081785	1332582004
214853381	89482149	602116046	593960624	1454735214
1306528138	62589812	1670073895	2126602119	396544957
359664239	1695987831	508747874	649683560	1355999881
1610666646	1438289155	2079645965	1136719287	611668365
1828435645	1878495191	1847615830	1237328482	1468948367
972970514	1010471390	126020853	706427382	1709121003
1101592370	1506604121	1512514978	824215950	1321742111
1888709996	995582175	1561995614	1304420267	1856996488
890285426	978341136	1151089329	2023442129	1385616379
1154363107	1700408166	942499575	1735119235	1443415006
968186004	857083457	1487296614	846315352	2084576224
389713771	614976766	1988625655	498035997	2091302020
884030647	563043968	1852865445	1605194785	1648446679
1259082738	1835824673	260994711	240447011	1147108816
15504300	1413287209	650491165	680854257	628062990
1435306322	927243288	1305700395	1434972591	593585162
1042208133	1894106185	1036034475	1673116415	241964875
434833139	843936096	1026031797	83283724	962428207
1766260325	522422173	153294169	544987095	885855294
1712991949	1186874058	1665120202	1481001151	1178284384
1300846825	100141984	1878222740	1194074653	1796829509
913776635	2047895118	584717610	1861041125	665538753
345309885	570635603	889304313	1370517560	1128124577
938844034	2062303428	1436861582	1786206933	1053597311
1580605747	1901854344	720358714	1236164771	2040072439

Multipliers for Prime Modulus $m = 2^{62} - 57$

2227666520522988	2335245043818421572	2958048408930287532	2821497825880930668
6125433456171879	229869021091622116	541962160899652940	3645820036832622356
242991343497935	2986508760285856708	364313872487046284	2117980445355000731
385325382108333	1230401549344097348	4437343137438423668	1077327186057395324
0445622603323716	3404718480622991346	3632484698526789692	3040394404002739493
4716400480585236	4425050684624664868	601696678466323924	3360988063228378564
3245363233199204	305480126024544988	1140520979900998343	2904568835753732236
910487642975572	3573939289300556044	1437104732804195596	1111204688074489044
5423650086846141	3176158344253306828	3665506303834541868	529750891505220676
0272513680219020	757998535101219594	982506319620817612	510301227247225548
7476194856468218	1476746311225538452	2339344678753297700	2406547870037429612
308330264249580	2028107709143540196	460465672511139916	2169322410029413884
6234903618635836	2836325001576828756	3304850999237510764	1302692003016593727
9507500607942393	1127184225209561420	2442328367503098604	3525967833252807900
3517735631097212	254533705499448356	3216739265730783020	2734845951024098108
9762161018925836	3143250031667232428	4312569826513122964	2858990330270135956
818812215302820	3505111250052047420	142682684051643767	2924706175868748540
9091477482083836	3005708043498447004	1122404151658602356	2384376175079003364
2101648724486855	3049914937445631084	2485606123337880225	3737901350162908743
3087629853475092	2820800426820749716	1803501089697376940	1322520419430537292
6375748891921916	771883998081009342	1514926988797966303	2213119349169260508
031080726473732	2243945721847679725	2930516100542645740	4494970451474388404
7462914720430958	3843182323716420380	2876990204610411756	1007531718334738490
8538904338130612	2606475706061829691	960695606911668116	2782767896275353612
7053581384735532	1643282104386383980	670108774451549844	1001393204888569492
8585308547598203	2418232767452531692	1621277272908803724	4008104371906474908
7809888395679756	1049643446522008484	432127321738792164	1789457316119133228
312008784666244	1849084705355803389	4261613820926935399	857519200068935628
0472151709439338	4067814427609890436	1273491878580529124	2083357865738529211
8098840758717236	3898454276482771700	2383006420636971996	447971768444903140
8097943894898732	2442060819726853316	1533254051312395243	468986501493944828
249454169311180	1188318551523486772	2593239100320542884	1683383931902134724

Multipliers for Prime Modulus $m = 2^{63} - 25$

5048131329874245129129	6970857753449530850	6420278701347409430	5124525484560574678
3599012081437622174	9033130285246985478	3135883546616849630	5450100294142127183
5230828488753169785	5277564623835390478	1243915603764351534	3067675794346700338
1164142900299891771	1690244713649970294	8504294721320585030	1982429857622105466
5863122408574880490	7488297927170703726	1192417868866640021	4674946439731140607
2161553242107985838	3693270853121744274	2723638313981663842	2583372058471374662
5583226340227159521	7440483159998091718	6865597922945174018	6710640246843790433
8130017674206960802	1355888514815117818	66935293183612951	7310597557252160706
2193035004633415366	698553738790551194	5359281404268310005	8724299514156051935
126418711700522650	3558263172428606031	6343443902471568174	4485294600469307766
7041410690023337563	7578608487980007799	253175814488353254	2347700846709651015
2138588662245218566	8408227919743527230	5206346059077753018	1955988046358844486
8189359621998424622	3616566817062986114	6424355341835960642	2500937059175260802
8080320067683804887	8033051016217168982	7321992163586838902	6411023698701546182
1769007136379980686	7783932335745702394	5558975998246609070	2143403440715322422
3179570434846579489	4945674464234686695	4695325265498138958	5288198982489587626
7762145188453129894	6245073051208444365	1478438406335434706	2309620102346487126
2287775950289931630	2457532858703768094	6610727231368630747	905762158313624865
870491490180669699	9197167186577252565	3870451705883074774	439605922369016570
4359681192108683725	643024978184937358	8162972527107683162	4667597220755320922
1937140893319420210	5487491960955870086	6672346787909502191	719372909398164950
3760870210522814318	1827366899773106907	8652243963382490303	5176898528264328250
1773166088840501934	2479698328010538454	9087242916220307793	2289446703802487557
400065671079016742	1035122893537674026	6561247231059429639	9169751783082340538
6564255993928279035	3997015511275500730	549512076501724846	1925625704083580246
844694661116743062	1456029575323840666	7410714248380123986	8688101763314670602
5595565586107992950	5611696577798111118	3927998942213401347	1561798619548458430
598858993587313666	8205566993320371494	7154943775958131514	5670532371588444962
9054079806224684046	7044308632905183510	7855708569000872254	4546508663771071157
2048336232069601622	192296947391005798	6940449059354001403	4595997938451814114
4832871382696361267	123352434606157734	676201409278475890	4217132583335612998
9077402637247533766	9035191738883420082	2228114199495086142	7909033251671519

Parallel Pseudorandom Number Generation

Recall, our primary interest is to compute pseudorandom numbers in parallel while avoiding the computation of any subsequence produced by another process. If every process used the same multiplier but a different randomly chosen seed, then it is possible multiple processes might produce the same subsequence. If each process uses a distinct carefully chosen (as described) multiplier , then we would not expect the same subsequence to be generated by two different processes, even though every pair of processes (eventually) generates the same set of numbers (in a different order). We shall discuss a couple different experiments.

For each experiment, we develop both MPI and hybrid MPI + OpenMP code. For the MPI version, we use 128 MPI processes on 32 nodes. For the hybrid version, we use 32 MPI processes on 32 nodes with 4 threads per node, which yields a total of 128 threads.

For all tests, we employ the same prime modulus $m = 8589934583 = 2^{33} - 9$, and the same initial seed $x_0 = 7927$, which is a primitive root of m. We chose this modulus because we were able to find ample multipliers with full period. Although our choice of a seed is arbitrary and some authors suggest picking a seed randomly, we would recommend choosing one randomly from a set of tested seeds because the period can be affected by the choice of seed for large moduli. Using an assigned seed and multiplier, each process or thread generates pseudorandom numbers in parallel as needed using Algorithm 2.

Each experiment consists of two parts. In the first part, each process or thread uses the same multiplier, namely 1178748639 (listed first in table below), and a different seed. We assign a seed to each process or thread uniquely according to its MPI rank and OpenMP thread number using a single **MCG** for all processes/threads

$$x_{n+1} = 66827594 \cdot x_n \bmod 2^{33} - 9,$$

which yields a full period and passes our empirical tests. In the MPI version, the process with rank r has seed x_r, whereas in the hybrid version the process with rank r and thread number t has seed x_{4r+t}. In the second part, each process or thread is assigned the same initial seed $x_0 = 7927$ and a unique multiplier from among 128 multipliers that pass our empirical tests and yield a full period, which we list below (the multiplier for the **MCG** above is not included).

Multipliers for $m = 8589934583 = 2^{33} - 9$

1178748639	1504451367	1438843832	1508982646
2039663820	1498584416	1753099100	1264049400
420951726	944773179	1894605422	1972037996
2145391955	1290515899	821331287	233089823
815719265	1037449050	391368831	1619611282
174343685	36512342	83440728	469601921
1887658788	409426010	710163604	720184858
642944870	165253310	2030930240	1015182040
1399937605	964731408	1370458820	928764526
835644228	964138849	771690904	2100758807
2058614442	1736063227	388665714	783217427
1981572859	297867486	1441424772	1649895453
1959842942	1855047552	1105029878	207427210
967163134	2073682087	159098215	1721887127
504646265	1483353159	1485525714	1538189469
1716278914	2033983412	2078532603	1926327885
1959643358	154598950	406575269	36921595
839205172	1494766337	1210100711	1837727164
1789532978	274166327	685199652	1183031368
2109888739	347221270	436678419	380986609
1171151118	1389829650	1912037874	324737772
1636694774	1508450362	1755102432	1619085308
617918706	1989869210	984783477	259413370
1316480107	1837397190	1240911029	197185544
830682663	1977680361	1403997604	291188364
1428272847	160535591	1093106588	1971097235
1899338587	574974465	707574314	743571448
787295652	2055016389	1786492583	447846030
614468563	198700516	627010989	1587207860
249082425	1132873853	717740897	901967693
1167409652	1363068688	1627762155	581967324
705969097	709166350	1570040924	429303093

Calculating Π in Parallel

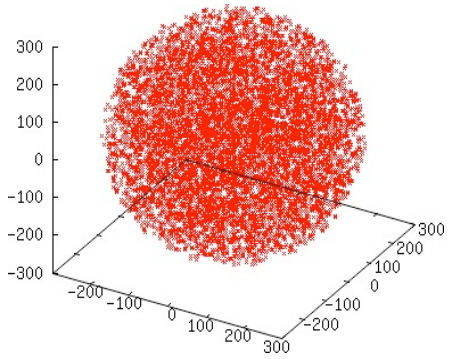
In the first experiment, we calculate π as described next. Pick $2^{32} = 1024^3$ points uniformly randomly from the cube whose edge has length 600 and count the number of points, say C, that coincide with the largest sphere that fits in the cube. The largest sphere has a diameter with the same length as an edge of the cube. Assuming a uniform distribution, then the number of points coincident with the sphere divided by the total number of points is approximately the quotient of the respective volumes, i.e.,

$$\frac{C}{2^{32}} \approx \frac{(4/3)\,\pi\,(300)^3}{600^3} = \frac{\pi}{6}.$$

So $\pi \approx 6C \cdot 2^{-32}$. We simulate picking points uniformly randomly by independently generating pseudorandom numbers for each coordinate. We summarize the results of our experiment below.

Test	Approximation to π MPI/Hybrid
same multiplier, different seed	3.1415769774466753
different multiplier, same seed	3.1415196494199336

Although we do not expect to see a difference between MPI and hybrid (except in terms of timing since even though the sum is computed differently, it is an integral sum), we would have expected greater accuracy when using different multipliers, which is not observed in this experiment. On the other hand, we do not lose a single digit of accuracy by using the same seed with different multipliers. To further check the quality of the distributions produced, we ran another test using MPI with different multipliers in which each MPI process generates a much smaller number of points, specifically 128 points per process, and collected all of the points that coincide with the sphere from all processes. The resulting graph displayed below indicates a satisfactory distribution.



3D Plot Using 128 MPI Processes with 128 Points per Process

Parallel Generation of Permutations

In the second experiment, we generate permutations in parallel. Given a sequence, $p_1, p_2, \ldots p_n$, a permutation is merely any reordering $p_{i_1}, p_{i_2}, \ldots p_{i_n}$. Permutations are useful for example when testing sorting implementations that depend on the input because it is impractical to exhaustively test every possible input. Yet even restricting the input to a single sequence (for instance, instead of testing all real numbers, an uncountable set, pick only 128 integers), it is impractical to test every permutation unless there are only around a dozen elements or less. For instance, a list with 128 elements can be rearranged in

128! = 38562048236258042173567706592346364061749310959022359027882840327637340257516554356068616858850736153403005183305891634759217293226249885776611495524503935776003464470927924769249558528000000000000000000000000000000

(216 decimal digits - unfortunately this number does not fit on one line) different ways, which astoundingly exceeds the number of hydrogen atoms in the observable universe [35]! Based on statistical theory, we know we need to generate only a relatively small number of permutations, provided the sample is chosen well.

There is a simple and fast way to generate a “pseudorandom permutation.” Choose any arbitrary permutation initially. Successively generate the next permutation using the previous permutation as follows. Uniformly randomly pick any position and swap the element in this position with the last element. After each swap the element in the rightmost position is in its final position. Recursively, uniformly

randomly pick a position corresponding to the elements excluding those elements in the tail that have already been placed and swap with the last element excluding the tail. Hence it takes only $n - 1$ swaps (and pseudorandom number generations) to generate the next permutation of a list with n elements. This method is amazing because it is not only fast but also produces permutations as if sampling from a uniform distribution [36]. In our tests, we found the generation of pseudorandom permutations (as described) seems to come from a uniform distribution even when the **MCG** used to swap elements fails many tests.

Our main interest is to generate permutations in parallel. Obviously two distinct processes should not generate the same permutation. Yet it is not enough to simply avoid the generation of the same permutations from among the incredibly vast number of them. Collectively, the permutations should appear as if they come from a uniform distribution. Ideally, this sampling is done independently without communication for the sake of speed.

In our experiment, each process (or thread) uses the same algorithm and an **MCG** as previously described to swap elements. Each process generates 20484 permutations of a list with 128 elements, starting with the same initial permutation which is arbitrarily chosen to be

{ 63,64,127,126,125,124,123,122,121,120,119,118,117,116,115,114,113,112,111,110,109,108,107,106,105,
104,103,102,101,100,99,98,97,96,95,94,93,92,91,90,89,88,87,86,85,84,83,82,81,80,79,78,77,76,75,74,73,
72,71,70,69,68,67,66,65,62,61,60,59,58,57,56,55,54,53,52,51,50,49,48,47,46,45,44,43,42,41,40,39,38,37,
36,35,34,33,32,31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0 }

We do not store and collect the actual permutations. Instead we store a single number to represent each permutation. There is a natural number to associate with each permutation. List all permutations in increasing lexicographical order and associate the position of each permutation in the ordered list with that permutation, e.g., {0,1,…,127} is assigned the number 0 and {127,126,…,0} the familiar number 128!. The only flaw with this numbering scheme is the numbers are unwieldy. Although sorting is really out of the question, figuring out the position is fast (much like looking up a phone number - if the first digit is k then we know the permutation comes after all permutations whose first digit is j with $j < k$, etc.). A solution to deal with the unwieldy numbers is to apply the same nice operation which is used to generate pseudorandom numbers themselves, i.e., modulo operation. Thus, after we generate a permutation we calculate its 'number' by computing its position using modular arithmetic, which is relatively costly in terms of operations compared to generating the permutation itself! We choose a reasonably large modulus M and precompute $factorial(k) = k! \bmod M$ for $k \in \{1,\ldots,n-1\}$, where n is the length of the list. We present our algorithm to compute the numbers assigned to a permutation next.

Algorithm 7: Map Permutation to an Integer ζ

Input: A permutation of $0,1,\ldots, n-1$, a modulus M, and $factorial(k) = k! \bmod M$, $1 \le k \le n-1$
Output: ζ, an integer between 0 and $M - 1$ (inclusive)

1. Put $\zeta \leftarrow 0$
2. Set $i \leftarrow 1$
3. While $i \neq n$ carry out steps a-e:
   a. Set current position $k \leftarrow (i+1)^{st}$ element from the right
   b. Put $e \leftarrow$ element at current position
   c. Let $C \leftarrow$ the number of elements preceding current position which are smaller than e
   d. Update $\zeta \leftarrow [\,((e - C)\cdot factorial(i)) \bmod M + \zeta\,] \bmod M$
   e. $i \leftarrow i + 1$

In our experiment we use the modulus $M = 2^{32}$ in the preceding algorithm because we easily obtain suitable data for the diehard tests. We collect all numbers computed for every permutation by all processes (threads) and print the numbers to a file. The resulting files were exactly the same for the MPI

and hybrid versions. We then ran diehard tests on these files. Nearly all of the diehard tests are clearly passed whether we use the same multiplier and different seeds or different multipliers and the same seed. Only a couple of the diehard tests are not clearly passed but the same failures occur regardless of the experiment (both parts). Although it is sufficient to use different seeds in this case, the results are also satisfactory using different multipliers.

Conclusion

We have described techniques to find and implement fast, high-quality customized parallel pseudorandom number/permutation generators. We reviewed the theoretical underpinnings. We gave a rigorous proof that there is a computational advantage choosing a prime moduli near a power of two. We demonstrated this advantage by simulation of throwing a die on the Cray XD1.

We have significantly improved the performance of an LLL reduction implementation and presented an algorithm to find many *candidate* multipliers for a prime modulus. We have shown that some multipliers with high LLL-spectral results (including some which are published [27]) are unsatisfactory. We also discovered an unexpected phenomenon for certain intermediate moduli: the vast majority of the multipliers with high LLL-spectral results are unsatisfactory based on empirical tests. We introduced a fast empirical testing procedure to select good multipliers from thousands of multipliers with satisfactory LLL-spectral results. We also introduced a modification of Brent's algorithm to quickly discover short periods, which we have shown is useful because the period is frequently short even when the LLL-spectral and empirical results are acceptable. We also introduced an algorithm to assign numbers to a permutation for statistical analysis.

It is not convincing to draw conclusions on the basis of a small number of experiments. More experiments are needed. Regardless, the only way to know whether a method of parallel number generation is adequate for an application is to conduct tests to assess the quality for each implementation. Scalability is a concern especially when using different seeds. We advocate using different multipliers chosen as described herein. Our experiments produced satisfactory results using different multipliers. For different runs we recommend using a different seed but rather than choosing a seed completely randomly, we recommend uniformly randomly picking a seed from a set that has been tested in advance.

For some moduli (not all such as $2^{39} - 7$) we found plenty of multipliers with both high LLL-spectral results and low ς statistics. It is unclear how to select multipliers from a large set of multipliers that will work best in parallel. It is impractical to exhaustively test all possible subsets with k elements (scaling to k processors) from a set of N multipliers (unless $k \approx N$), e.g., the number of subsets having 128 multipliers from a set of 1024 multipliers has 167 digits.

Acknowledgements

This work was performed entirely on the Cray XD1 system at NRL-DC under the auspices of the U. S. Department of Defense (DoD) High Performance Computer Modernization Program (HPCMP).

References

[1] Knuth, D.E. **The art of computer programming, Volume 2 (3rd ed.): seminumerical algorithms**, Addison-Wesley Longman Publishing Co., Inc., Reading, Massachusetts, 1997.

[2] Jacobs, K. **Invitation to Mathematics**, Princeton University Press, 1992.

[3] Dickson, L. **History of the Theory of Numbers**, Published by AMS Bookstore, 1999.

[4] Chen, W. **Elementary Number Theory**,http://www.maths.mq.edu.au/~wchen/lnentfolder/ent03-c.pdf, 2003.

[5] Pickover, C. **The Möbius Strip: Dr. August Möbius's Marvelous Band in Mathematics**, Games, Literature, Art, Technology, and Cosmology, Thunder's Mouth Press, 2006.

[6] GNU Project **The GNU MP Bignum Library**, http://gmplib.org/, 2008.

[7] Shoup, V. **NTL: A Library for doing Number Theory**, http://www.shoup.net/ntl/, 2008.

[8] Caldwell, C. **The Top Twenty Sophie Germain Primes**, http://primes.utm.edu/top20/page.php?id=2, 2009.

[9] Mascagni, M. and Hongmei, C. **Parallel linear congruential generators with Sophie–Germain moduli**, Parallel Computing, Volume 30, Issue 11, 2004.
[10] Mascagni, M. **Parallel linear congruential generators with prime moduli**, Parallel Computing, Volume 24, 1998.
[11] De Matteis, A. and Pagnutti, S. **Parallelization of Random Number Generators and Long-Range Correlations**, Numerische Mathematik, Volume 53, 595 608, 1988.
[12] Wagstaff, S. **Cryptanalysis of Number Theoretic Ciphers**, CRC Press, 2003.
[13] McMath, S. **Daniel Shanks' Square Forms Factorization**, http://www.usna.edu/Users /math/wdj/mcmath/SQUFOF.pdf, 2004.
[14] Rabin, M. **Probabilistic Algorithm for Testing Primality**, Journal of Number Theory, Volume 12, pp. 128-138, 1980.
[15] L'Ecuyer, P. and Andres. T. **A random number generator based on the combination of four LCGs**, Mathematics and Computers in Simulation, Volume 44, pp. 99-107, 1997.
[16] Sezgin, F. **A method of systematic search for optimal multipliers in congruential random number generators**, BIT, Volume 44, Number 1, pp. 135-149, 2004.
[17] Borosh, I. and Niederreiter, H. **Optimal multipliers for pseudo-random number generation by the linear congruential method**, BIT, Volume 23, pp. 65-74, 1983.
[18] Mascagni, M. and Srinivasan, A. **Algorithm 806: SPRNG: a scalable library for pseudorandom number generation**, ACM Transactions on Mathematical Software (TOMS), Volume 26 , Issue 3, 2000. pp. 436-461 , 1998.
[19] Hellekalek, P. **Good random number generators are (not so) easy to find**, Mathematics and Computers in Simulation, Volume 46, pp. 485-505, 1998.
[20] Coddington, P. **Random number generators for parallel computers**, The NHSE Review (2), 1996.
[21] The GSL Team **The GNU Scientific Library - a free numerical library licensed under the GNU GPL**, http://www.gnu.org/software/gsl/manual/html_node/Other-random-number-generators.html, 2008.
[22] Matthews, K. **Finding the least primitive root (mod p), p an odd prime**, http://www.numbertheory.org/php/lprimroot.html, 2004.
[23] Entacher, K. **On the CRAY-System Random Number Generator**, Simulation, Volume 72, Number 3, pp. 163-169, 1999.
[24] Lenstra, A., Lenstra, H. and Lovasz, L. **Factoring Polynomials with Rational Coefficients**, Math. Ann. 261, pp. 515-534, 1982.
[25] L'Ecuyer, P. and Couture, R. **An Implementation of the Lattice and Spectral Tests for Multiple Recursive Linear Random Number Generators**, INFORMS Journal on Computing, Volume 9, pp. 206-217, 1997.
[26] van der Kallen, W. **Complexity of an extended lattice reduction algorithm**, http://www.math.ruu.nl/people/vdkallen/kallen.html/complexity.ps, 1998.
[27] Entacher, K. **Efficient lattice assessment for LCG and GLP parameter searches**, Mathematics of Computation, Volume 71, Number 239, pp. 1231-1242, 2001.
[28] Entacher, K. **Spectral Test Server**, http://random.mat.sbg.ac.at/results/karl/spectraltest /index.html, 2000.
[29] Simmons, G. **Calculus Gems**, McGraw Hill Inc., 1992.
[30] Nivasch, G. **Cycle detection using a stack**, http://www.yucs.org/~gnivasch/stackalg /index.html, pp. 135-140, 2004.
[31] Nivasch, G. **The Cycle Detection Problem and the Stack Algorithm**, http://www.yucs.org /~gnivasch/stackalg/index.html, 2004.
[32] Brent, R. **An improved monte carlo factorization algorithm**, BIT, Volume 20, pp. 176-184, 1980.
[33] Marsaglia, G. **The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness**, http://www.stat.fsu.edu/pub/diehard/, 1995.
[34] Thornton, T., and McPeek, M. **Case-Control Association Testing with Related Individuals: A More Powerful Quasi-Likelihood Score Test**, American Journal of Human Genetics, Volume 81, pp. 321-337, 2007.
[35] Champion, M. **How many atoms make up the universe?**, http://www.madsci.org/posts

/archives/oct98/905633072.As.r.html, 1998.
[36] Diaconis, P. and Shahshahani, M. **Generating a random permutation with random transpositions**, Probability Theory and Related Fields, Volume 57, Number 2, pp. 159-179, 1981.

About the Authors

Stephen Bique is Computer Scientist in the Center for Computational Science (CCS) at the Naval Research Laboratory (NRL). Robert Rosenberg is Information Technology Specialist in CCS at NRL.