PARALLEL LINEAR CONGRUENTIAL GENERATORS WITH PRIME MODULI

Michael Mascagni

Program in Scientific Computing
and
Department of Mathematics
The University of Southern Mississippi

Abstract. Linear congruential generators (LCGs) remain the most popular method of pseudorandom number generation on digital computers. Ease of implementation has favored implementing LCGs with power-of-two moduli. However, prime modulus LCGs are superior in quality to power-of-two modulus LCGs, and the use of a Mersenne prime minimizes the computational cost of generation. When implemented for parallel computation, quality becomes an even more compelling issue. We use a full-period exponential sum as the measure of stream independence and present a method for producing provably independent streams of LCGs in parallel by utilizing an explicit parameterization of all of the primitive elements modulo a given prime. The minimization of this measure of independence further motivates an algorithm required in the explicit parameterization. We describe and analyze this algorithm and describe its use in a parallel LCG package.

1. Introduction.

Perhaps the oldest generator still in use for the generation of uniformly distributed integers is the linear congruential generator (LCG). This generator is sometimes called the "Lehmer" generator, in honor of its originator, D. H. Lehmer, the father of electronic computational number theory, [13]. The LCG is based on the following modular integer recursion for producing pseudorandom integers:

$$x_n = ax_{n-1} + b \pmod{m}. \tag{1}$$

Equation (1) defines a sequence of integers modulo m starting with x_0, the initial seed. The constants of the recursion are referred to as the *modulus* m, *multiplier* a, and *additive constant* b.

Since LCGs are so commonly used as serial pseudorandom generators, we feel that a useful method for the implementation of LCGs on parallel machines is required. There has been some work on the splitting of full-period LCG sequences into shorter subsequences for use in parallel, [7, 5]. This paper takes an altogether different approach to parallelizing LCGs. We seek a parameterization of complete

1991 *Mathematics Subject Classification.* 65C10, 65Y05.

Key words and phrases. pseudorandom number generation, parallel computing, linear congruential generators.

and distinct full-period LCG sequences so that each new parallel process can use an entirely distinct full-period sequence. To our knowledge this has been examined by only one group, [18], where the parameterization of power-of-two modulus LCGs was studied by varying the additive constant, b, in the recursion (1). In this paper we will study the consequences of parameterizing full-period LCG sequences when the modulus, m, is prime.

The plan of the paper is as follows. In §2 we will review some well known results from the theory of LCGs. The motivation for this is to set up the mathematics of LCGs and convince the reader that prime modulus LCGs offer some compelling advantages over power-of-two modulus LCGs. In §3 we decide upon the use of the multiplier as the means of parameterizing prime modulus LCGs. We then describe an explicit enumeration of all of the distinct full-period cycles for a prime modulus LCG based on this parameterization. In §4 we present a result from number theory that gives a qualitative measure of the full-period correlation of different sequences parameterized in this way. This result, based on the Riemann hypothesis over finite fields, also provides a heuristic for choosing the parameters. In §5 we study an algorithm that implements this heuristic based on computing the kth integer relatively prime to a given, factored, integer. In §6 we briefly describe a mapping of parallel processes onto a binary tree to provide a very versatile parallelization. In addition, we describe the state of a package we have written that implements these ideas for parallel LCGs. Finally, in §7 we give our conclusions and comment on directions for future work.

2. Linear Congruential Generators.

Equation (1) yields a perfectly periodic sequence with period defined by the seed and m, a, and b. We refer to the period of the sequence $\{x_n\}$ as $\mathrm{Per}(x_n)$. When m is prime, $\mathrm{Per}(x_n) = m - 1$ is the longest period achievable, occurring when a is a primitive element modulo m, [9].[1] With a primitive modulo m, any choice for b gives $\mathrm{Per}(x_n) = m-1$. For this reason it is customary to choose $b = 0$, since the set of $m - 1$ elements in the full period of (1) will contain all the residues in $\mathbb{Z}/(m)\mathbb{Z}$ except the x that satisfies the equation $x = ax + b \pmod m$.

When $m = 2^k$ the question of longest possible period falls into two cases. With $b = 0$, the largest value of $\mathrm{Per}(x_n)$ is 2^{k-2} when either $a \equiv 3$ or $5 \pmod 8$, [9]. When $b \neq 0$ the largest value of $\mathrm{Per}(x_n)$ is 2^k. This occurs when b is odd and $a \equiv 1 \pmod 4$, [9]. Thus we see that the period size of LCGs modulo a power-of-two can be as large as the number of residues modulo 2^k and hence is comparable to the period of LCGs modulo a prime. A major shortcoming of LCGs modulo a power-of-two compared with prime modulus LCGs derives from the following theorem for LCGs, [9]:

Theorem 2.1. *Define the following LCG sequence:* $x_n = ax_{n-1} + b \pmod{m_1}$. *If* m_2 *divides* m_1 *then* $y_n = x_n \pmod{m_2}$ *satisfies* $y_n = ay_{n-1} + b \pmod{m_2}$.[2]

To understand the consequences of Theorem 2.1 consider a maximal period LCG with $\mathrm{Per}(x_n) = m = m_1 = 2^k$ and let $m_2 = 2^j, 0 < j < k$. Forming $y_n = x_n$

[1] We say that a is a primitive element modulo m if the powers of a modulo m take on the value of every positive residue modulo m. More exactly, with m prime, the residue a is primitive if and only if the set $\mathbb{A} = \{x | x = a^i \pmod m, 1 \leq i \leq m-1\} \equiv \mathbb{Z}/(m)\mathbb{Z}^+$.

[2] Theorem 2.1 actually holds for all linear modular recursions.

(mod m_2) is just taking the j least-significant bits of the LCG. Theorem 2.1 then implies that the j least-significant bits of any power-of-two modulus LCG with $\mathrm{Per}(x_n) = 2^k$ has $\mathrm{Per}(y_n) = 2^j, 0 < j < k$. In this case the least-significant-bit of the LCG has period 2, the two least-significant-bits have period 4 and so on. Since a long period is often thought of as a partial remedy for determinism in pseudorandom number generators, when these types of LCGs are employed in a manner that makes use of only a few least-significant-bits their quality may be seriously compromised. When m is prime no such problems arise.

The costliest computational task when iterating (1) is the modular arithmetic. Modular addition is comparable in cost to integer addition; however, modular multiplication can be much more expensive than plain integer multiplication. When the modulus is a power-of-two, i.e., $m = 2^k$ with $k > 0$, the cost of modular and regular multiplication is comparable. In fact, if k-bit integer multiplication hardware is used, the costs are identical. We have seen that there are theoretical reasons why using a prime number for m is optimal. Because of this state of affairs the only moduli that have been used in practical implementations are $m = 2^k$ or m a Mersenne prime, i.e., a prime of the form $2^p - 1$.[3] With a Mersenne prime, modular multiplication can be implemented by performing the full integer multiplication with only the inclusion of bitwise shifting and integer addition required to accomplish the modular reduction, [1]. The reader will be convinced by considering the relationship bewteen modular redution modulo 2^p and $2^p - 1$. Thus in the sequel we will focus on Mersenne prime modulus LCGs to minimize the cost of modular multiplication for an efficient implementation.

3. Parameterization via the Multiplier.

There are many ways to parallelize recursions used in serial pseudorandom number generators. One method is to split the full-period sequence into subsequences that are then used on individual parallel processors, [5]. Recently, certain pseudorandom number generators have been parallelized using different seeds to select different full-period cycles, [15, 19]. This is a form of parameterization of the full-period cycle. Another form of parameterization has been used on LCGs with power-of-two moduli, [18]. Here a different additive constant was used to produce different LCG sequences. We feel that this parameterization is an intriguing approach that has yet to be applied to prime modulus LCGs.

To parameterize a prime modulus LCG one can vary either the modulus, the multiplier or the additive constant. We feel that it is unacceptable to vary the modulus. The number theoretic properties of this modulus are used to optimize the modular multiplication. Thus, using a different modulus on different parallel processes will lead to pseudorandom number generation codes with very different execution times per pseudorandom number. Another compelling reason to avoid considering modular parameterization is that the theoretical measure of interprocessor correlation we use later in this paper is analytically intractable with different moduli.

[3] It is true that Fermat primes, primes of the form $2^{2^n} + 1$, have similarly efficient modular multiplication routines. However, there are substaintially fewer Fermat primes than Mersenne primes. Because of this fact we only consider Mersenne primes in this paper.

Having eliminated the modulus from consideration we have the choice of parameterizing the additive constant or the multiplier. We have chosen to parameterize the multiplier for a variety of reasons. One of the most compelling is as follows. Let $x_n = ax_{n-1} + b_x \pmod m$ and $y_n = ay_{n-1} + b_y \pmod m$. Equivalently we can write $x_n = a^n x_0 + b\frac{a^n-1}{a-1} \pmod m$ and $y_n = a^n y_0 + b_y\frac{a^n-1}{a-1} \pmod m$ provided that $\gcd(a-1, b_x) = \gcd(a-1, b_y) = 1$. Some algebra gives:

$$(2)\quad x_n - y_n = a^n(x_0 - y_0) + (b_x - b_y)\frac{a^n - 1}{a-1} = a^n\left[(x_0 - y_0) + \frac{b_x - b_y}{a-1}\right] + \frac{b_y - b_x}{a-1}.$$

Since x_0 and y_0 are arbitrary positive integers modulo m, it is fair to explore all possible starting values for y_0 with x_0 given. Obviously if we choose $x_0 = y_0 + \frac{b_y - b_x}{a-1}$ then this gives $x_n - y_n = \frac{b_y - b_x}{a-1}$, a constant. Thus given any pair of prime modulus LCGs with the same multiplier, there is a set of initial conditions that makes their difference a constant! Obviously this leads to pseudorandom numbers that are extremely correlated.

A further consideration in favor of parameterizing the multiplier over the additive constant is that when parameterizing the multiplier we may choose all the additive constants to be zero. This further speeds the implementation, as only one modular multiplication and no modular addition is required per pseudorandom number.

Our desire is to determine an effective parameterization of the full-period prime modulus LCG sequences. Since we are only parameterizing the multiplier, as mentioned above, we can set $b = 0$. Recall that the conditions for an LCG of the form $x_n = ax_{n-1} \pmod m$ to have the maximal period is that $x_0 \neq 0$ and that a must be primitive modulo m. Thus if we can parameterize all of the primitive elements modulo m we will also have parameterization of all of the full-period LCG sequences modulo m. A useful theorem in this regard is:

Theorem 3.1. *If a and α are primitive elements modulo the prime, m, then $\alpha = a^\ell \pmod m$, where $\gcd(m-1, \ell) = 1$.*

This gives us a parameterization as follows. Let $\pi^{-1}_{m-1}(k)$ denote the kth number relatively prime to $m-1$. This notation will be justified later. Then we can define the kth primitive element as $a_k = a^{\pi^{-1}_{m-1}(k)} \pmod m$ where $a_1 = a^1 = a$ is known to be primitive modulo m. This observation reduces the parameterization to an explicit computation of $\pi^{-1}_{m-1}(k)$, the kth number relatively prime to $m-1$. In cases where $m-1$ has an explicit factorization, i. e., when $m = 2^{2^n} + 1$ is Fermat or when $m = 2q + 1$ with q prime is Sophie-Germain, one can write down $\pi^{-1}_{m-1}(k)$ explicitly. Since we are interested in Mersenne moduli, we have no such luxury and must consider the general case. We will do this in great detail after we first consider the calculation of a theoretical measure of interprocessor correlation that also further motivates the need to compute $\pi^{-1}_{m-1}(k)$.

4. Exponential Sum Cross-correlations.

A very common theoretical measure of the quality of a serial pseudorandom number generator is a metrical quantity known as the discrepancy, [10, 14, 16, 17]. The discrepancy of a sequence measures its equidistribution quantitatively by computing the maximal deviation of the given sequence from the uniform distribution.

This equidistribution test is commonly called the serial test, and can be done in any dimension and with either the full-period sequence or only a partial-period subsequence. When the pseudorandom number sequence comes from a recursion, one can often bound the discrepancy in question above and below with exponential sums.

Exponential sums are of interest in many areas of number theory. We define the exponential sum for the sequence of residues modulo m, $\{x_n\}_{n=0}^{k-1}$, as:

$$C(k) = \sum_{n=0}^{k-1} e^{\frac{2\pi i}{m} x_n}. \tag{3}$$

If the x_n are periodic and $k = \mathrm{Per}(x_n)$, the (3) is called a full-period exponential sum. If x_n is periodic and $k < \mathrm{Per}(x_n)$, then (3) is a partial-period exponential sum. Examining (3) shows it to be a sum of k quantities on the unit circle. A trivial upper bound is thus $|C(k)| \leq k$. If the sequence $\{x_n\}$ is indeed uniformly distributed, then we would expect $|C(k)| = O(\sqrt{k})$, [10]. Thus the desire is to show that exponential sums of interest are neither too big nor too small to reassure us that the sequence in question is theoretically equidistributed.

Since we are interested in studying sequences for use in parallel, we must consider the cross-correlations among the sequences to be used on different processors. If $\{x_n\}$ and $\{y_n\}$ are two sequences of interest then their exponential sum cross-correlation is given by:

$$C(i, j, k) = \sum_{n=0}^{k-1} e^{\frac{2\pi i}{m}(x_{i+n} - y_{j+n})}. \tag{4}$$

Here the sum has k terms and we start with x_i and y_j.

In a previous work we only considered full-period exponential sum cross-correlation for studying these issues for a different recursion, [19]. We will take the same approach here. Thus we are interested in studying full-period exponential sum cross-correlations when $x_n = ax_{n-1} \pmod m$ and $y_n = \alpha y_{n-1} \pmod m$ with $\alpha = a^\ell \pmod m$, $gcd(\phi(m), a-1) = 1$. Since there is no additive constant, both $\{x_n\}$ and $\{y_n\}$ omit only zero in their full period, so there is a positive residue modulo m, z, and an index n such that $x_n = y_n = z$. Without loss of generality let $x_0 = y_0 = z$ so $x_n = a^n z \pmod m$ and $y_n = a^{n\ell} z \pmod m$. This allows us to rewrite the difference in the summand of (4) as $f(a^n) = z\left(a^n - (a^n)^\ell\right)$ with $f(x) = z(x - x^\ell)$. Thus we can rewrite (4) as:

$$C(\cdot, \mathrm{Per}(x_n)) = \sum_{n=0}^{\mathrm{Per}(x_n)-1} e^{\frac{2\pi i}{m} f(a^n)} = \sum_{n=0}^{\mathrm{Per}(x_n)-1} e^{\frac{2\pi i}{m} f(n)}. \tag{5}$$

We are permitted to replace $f(a^n)$ by $f(n)$ in the full-period sum since the values of a^n run over the same range as n with a primitive modulo m.

Exponential sums that range over all the values of polynomials modulo m were studied by André Weil, and their bounds constitute some of the consequences of

the Riemann Hypothesis over finite fields. In particular, sums of the form (5) are known to satisfy

$$|C(\cdot, \mathrm{Per}(x_n))| \leq (\ell - 1)\sqrt{m} = O(\sqrt{m}), \tag{6}$$

[20]. Suppose we have j full-period LCGs defined by $x_{k_n} = a^{\ell_i} x_{k_{n-1}} \pmod m$, $0 \leq i < j$. All of the pairwise full-period exponential sum cross-correlations will satisfy

$$|C(\cdot, \mathrm{Per}(x_n))| \leq (\left[\max_k \ell_k\right] - 1)\sqrt{m}. \tag{7}$$

This inequality is minimized if $\ell_k = \pi_{m-1}^{-1}(k)$ and further motivates the need for an efficient algorithm to compute this function.

5. Computing $\pi_{m-1}^{-1}(k)$.

We are interested in computing the kth number relatively prime to a given number. We have chosen this notation to be similar to that of the number theoretic function $\pi(x)$, the number of primes less than or equal to x. The inverse of $\pi(x)$ gives us the kth prime since if $\pi(x) = k$ then $\pi^{-1}(k) = x$. Similarly, we count the number of integers less than or equal to x relatively prime to $m-1$ with the function $\pi_{m-1}(x)$, and so if $\pi_{m-1}(x) = k$ then $x = \pi_{m-1}^{-1}(k)$ is the kth number relatively prime to $m-1$. In the subsequent discussion on computing $\pi_{m-1}^{-1}(k)$ we take advantage of the previous algorithmic work for computing $\pi(x)$ for large values of x, [4, 11].

In our application, computing the kth primitive element modulo the prime m, we need to compute the kth number relatively prime to $\phi(m) = m-1$, when m is prime. We will assume that for the particular prime, m, $m-1$ has a known factorization. For example, if m is Mersenne, i.e., $m = 2^p - 1$, then $m - 1 = 2(2^{p-1} - 1)$. It is well known that p must be prime when $m = 2^p - 1$ is prime, and so we can assume that p is odd. If not, $p = 2$, and in that case $m = 3$, which is an unsuitably small modulus. Thus with p odd we may write $2^{p-1} - 1 = (2^{(p-1)/2} + 1)(2^{(p-1)/2} - 1)$. Using a notation consistent with the factorization tables of the Cunningham Project we write $(2^{(p-1)/2} + 1)(2^{(p-1)/2} - 1) = (2+)[(p-1)/2] \times (2-)[(p-1)/2]$. Complete factorizations of the integers $(2+)[(p-1)/2]$ and $(2-)[(p-1)/2]$ can then found in the Cunningham Project Tables for all values of p that are reasonable for implementation, [3].

Before we begin describing our algorithm let us fix some notation. First let us denote $\mathbb{B} = \{primep, \text{ such that } p|(m-1)\}$ as the set of all prime factors of $m-1$. Also we have $b = |\mathbb{B}|$ defined as the number of distinct prime factors of $m-1$. In addition we write the prime factors of $m-1$ as $p_1 < p_2 < \cdots < p_b$. We compute $\pi_{m-1}^{-1}(k)$ by an iterative search, progressing to a linear search once a certain threshold is reached. Our algorithm begins with a very educated guess, x^*, for a starting value for $x = \pi_{m-1}^{-1}(k)$. We then compute $\pi_{m-1}(x^*) = k^*$. We then iteratively refine this guess. In the sequel we will refer to x^* as the current guess for x and k^* as the current guess for k.

I. Starting value. Assuming that numbers relatively prime to $m-1$ are uniformly distributed, we expect that $\pi_{m-1}^{-1}(k) \approx \frac{k(m-1)}{\phi(m-1)}$. Thus we initialize with the guess:

$$x^* := \frac{k(m-1)}{\phi(m-1)}. \tag{8}$$

Although this value is inexact, it is an excellent first approximation to the correct result.

II. Computing $\pi_{m-1}(x^*)$. We next compute $\pi_{m-1}(x^*)$, the number of integers less than or equal to x^* and relatively prime to $m-1$, by the method of inclusion/exclusion. There are x^* integers less than or equal to x^*, and $\lfloor x^*/p_i \rfloor$ of them are multiples of p_i. Thus we can approximate $\pi_{m-1}(x^*)$ by subtracting these integers from x^*. However, we have over compensated since we have subtracted off the contributions from integers that are multiples of more than one of the primes more than necessary. We correct that by adding back $\lfloor x^*/(p_i p_j) \rfloor$. Again we must correct by adding back the contributions of three prime factors at a time, and so on. Formally we compute the desired quantity via the finite sum:

$$\pi_{m-1}(x^*) = x^* + \sum_{n=1}^{b} \left[(-1)^n \sum_{D \in \mathbb{D}_n} \left\lfloor \frac{x^*}{D} \right\rfloor \right] = \sum_{D \in \mathbb{D}} \left\lfloor \frac{x^*}{D} \right\rfloor. \tag{9}$$

Here $\mathbb{D}_n$ is the set of all products of n distinct prime factors of $m-1$ that appear as the denominators in the above formula. This is a sum of 2^b terms. For computational reasons it is convenient to also write (9) as a single sum by introducing the set of all signed denominators $\mathbb{D}$. Note that $1 \in \mathbb{D}$ corresponds to the first term in the first sum in (9) and the $2^b = |\mathbb{D}|$ elements of $\mathbb{D}$ are distinct from unique factorization.

III. Computing $\mathbb{D}$. We can compute the set $\mathbb{D}$ via a simple recursive algorithm. Since generally we will be computing $\pi_{m-1}(\cdot)$ for several values of the argument it is expeditious to amortize the cost of computing $\mathbb{D}$ over all these evaluations. Thus we assume that we have an upper bound on the largest value of $\pi_{m-1}(\cdot)$ to be computed. Call that bound G. The recursive algorithm proceeds as follows. If there are no prime factors of $m-1$, then clearly $\pi_{m-1}(x^*) = x^*$ and $\mathbb{D} = \{1\}$. Otherwise, we remove the smallest prime factor of $m-1$, call it p, and recursively evaluate $\mathbb{D}$ for $(m-1)/p$. Call this new set $\mathbb{D}^p$. Obviously $\mathbb{D} = \mathbb{D}^p \bigcup -p \times \mathbb{D}^p$. We can use G in this recursive definition as follows. Since we compute the elements of $\mathbb{D}$ in increasing absolute value, we may terminate any recursive step when the next computed absolute value exceeds G. This guarantees the absolute minimum memory is utilized.

IV. Iterative search. Since $\pi(\cdot)$ is an increasing, integer-valued function, we may evaluate $\pi_{m-1}^{-1}(\cdot)$ by using our current guess, x^*, as follows. First evaluate $k^* = \pi_{m-1}(x^*)$ as described in step **III**. If the result is close enough to k, then progress with a linear search (see step **V**). Otherwise, update the guess with:

$$x^* := x^* + c\pi_{m-1}(x^*). \tag{10}$$

Here $c = \pi_{m-1}^{-1}(1)$ can be precomputed and is exactly equal to the smallest prime not in $\mathbb{B}$. This iterative search is rather slow because of repeated evaluation of $\pi_{m-1}(\cdot)$. To accelerate this procedure we try to minimize these expensive function evaluations by setting a rather high threshold for progressing to the linear search, i.e., 50,000. In other words if $|k - k^*| > 50,000$ we iterate, otherwise we proceed to step **V**, linear search. This ensures that $\pi_{m-1}(\cdot)$ is rarely evaluated more than two or three times in practice. This fortuitous behavior is due in large part to the accuracy of the initial guess.

V. Linear search. A linear search can be implemented quickly and effectively by divisibility testing. Starting with x^* and k^*, we add or subtract one from x^* (depending on whether k^* larger or smaller than k) and then test the new x^* for divisibility by the prime factors of $m-1$. If it is divisible by one or more, continue the search. If not, increment or decrement k^* and decide whether to continue. The search stops when $k^* = k$ and $\pi_{m-1}(x^* - 1) < k$. While it is convenient to check divisibility of x^* with each factor, it is conceptually nicer to compute $\gcd(x^*, m-1)$. While theoretically less costly than trial division, we found little difference in our implementation, and so we have continued to use trial division.

Optimizations. The denominators, $\mathbb{D}$, can consume large amounts of memory when $m-1$ has many large factors. In order to reduce the memory requirements of this algorithm, we apply the recursive formula:

$$\pi_{m-1}(x) = \pi_{(m-1)/p}(x) - \pi_{(m-1)/p}(x/p).$$

This allows us to reduce the number of denominators stored. We remove the smallest primes first, for the most efficient memory reduction when computing with the upper bound, G.

6. Parallelization and Implementation.

In the previous section we described an algorithm for computing $\pi_{m-1}^{-1}(k)$. Given this capability we have a parameterization of the full-period LCG sequences modulo m by associating the ℓth parallel process with the LCG $x_n = a_\ell x_{n-1} \pmod m$. Here $a_\ell = a^{k_\ell} \pmod m$ with $k_\ell = \pi_{m-1}^{-1}(\ell)$ and a is primitive modulo m. With m prime, there are at most $\phi(m-1)$ distinct primitive elements that can serve as multipliers, so this method provides at most this number of full-period LCGs. Given $m-1$ and its factorization one can compute $\phi(m-1)$ explicitly, [8]. However, it is more generally known that $\phi(m-1) \approx m/\log_2 \log_2(m)$. We have implemented this algorithm as part of a parallel linear congruential generation package in a portable manner using the GNU project's multiprecision package `gmp`. We have taken special care to optimize this code to take advantage of the division free modular reduction for Mersenne primes.

In our research on parallel pseudorandom number generation we have set four criteria which we require of any parallel pseudorandom number generator as our point of departure for investigations. These four criteria are:

(i) The generator must be able to provide a totally reproducible stream of parallel pseudorandom numbers. (This reproducibility must hold independent of the number of processors used in the computation and of the loading produced by sharing of the parallel computer.)

(ii) The generator must allow for the creation of unique pseudorandom number streams on a parallel machine without any interprocessor communication.

(iii) The generator must be portable between serial and parallel platforms and available on the most commonly used workstations and supercomputers.

(iv) The generator must provide "high quality" pseudorandom numbers in a computationally inexpensive and scalable manner.

The difficulty of this problem can be seen by considering the example of Monte Carlo applied to a problem in neutronics, [21]. Here independent neutron paths are generated based on the outcome of many events whose probabilities are understood. Statistics are collected along the paths, and computation produces estimates for quantities of interest that have a standard error that decreases as $N^{-1/2}$. (Here N is the number of "independent neutron paths.") The computational catch is that during flight a neutron may collide with a heavy nucleus, thereby producing new neutrons. These new neutrons, along with their initial conditions, are put into a computational queue for later processing. An efficient parallel implementation demands this queue be distributed. This complicates criterion **(i)** by implying that each neutron in the queue must also have information so that a unique and deterministic stream of pseudorandom numbers will be used regardless of which processors it eventually executes on. In addition, we wish the pseudorandom number streams that are allocated for one neutron to be distinct from those used for another neutron. Criterion **(ii)**, which forbids interprocessor communication in the course of assigning unique pseudorandom number streams, requires that the generator be flexible enough to assign distinct streams independent of the streams assigned elsewhere in the computation on any processor. Thus criteria **(i)** and **(ii)** and the demands of neutronics lead to very substantial and specific demands on any generator we would consider acceptable.

Criterion **(iii)** along with **(i)** imply that one can use a generator with these properties to obtain identical streams of pseudorandom numbers on different serial and parallel machines, including networks of workstations. This is essential to check codes ported to new platforms and to check the consistency of calculations when a hardware error may have been detected during a long run. Finally, criterion **(iv)** asks that the pseudorandom numbers be effective at the desired variance reduction, which is, of course, the point of large Monte Carlo calculations.

When one has a parameterization of the full-period cycles of a pseudorandom number generator that satisfies criterion **(iv)**, there is a canonical technique for achieving the other criteria. The Weil bound, equation (6), shows that this parameterization of LCGs satisfies criterion **(iv)**, so what remains is a description of the canonical technique. In a previous article, [19], the author describes a canonical technique for mapping a large number of parameterized parallel pseudorandom number generators onto a binary tree to permit an efficient, portable, and reproducible MIMD implementation. The point of using a binary tree to map the parallel processes is that one defines an entire subtree with each assignment and insures that processes elsewhere in the computation cannot accidentally assign the same process. In addition, the computation of what node and subtree follow can be done with only local information.

More details of this enumeration can be found in [19]. However, many of these details can be understood by working through a small example. In Figure 1 we

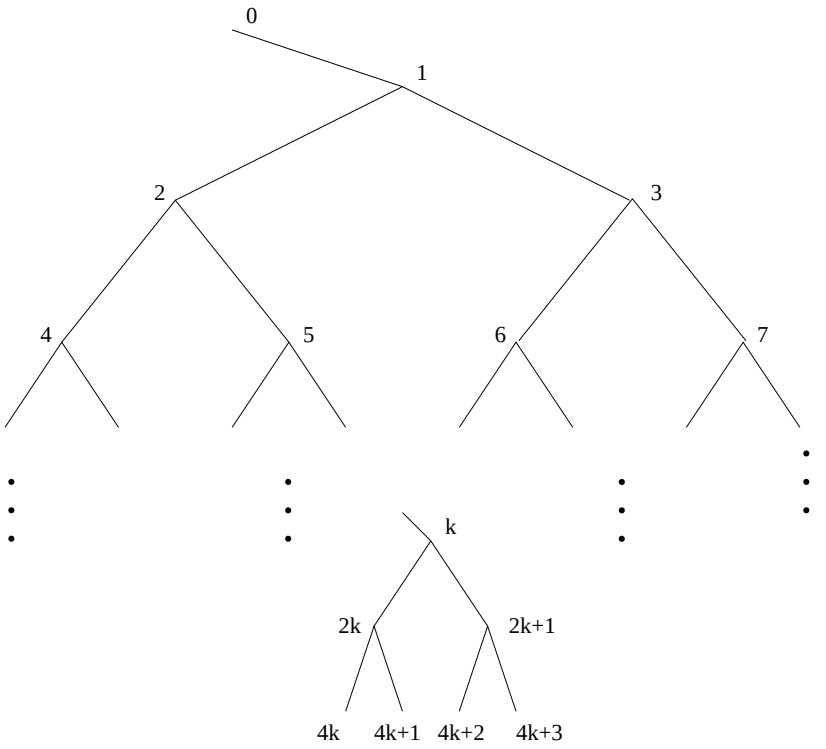

FIGURE 1. Enumeration of a binary tree.

show the canonical enumeration of nodes with integers designating nodes on the binary tree. We will refer to the parameterized generators by these same indices. A simple data structure is required to manage the algorithm. This date structure contains both the node of binary tree corresponding to the processor as well as the node where the first child process is to be assigned. We denote this as the tuple $(n, c(n))$ where generator n is pointing to $c(n)$ as the location of its first child. Suppose it is determined that the process pointing to generator q needs to spawn r new processes, each with its own generator. Each new generator must be different from the other new ones and different from any that some other existing generator may spawn. The new tree nodes assigned in this operation are:

$$q, 2q, 2q+1, 4q, 4q+1, 4q+2, \ldots, q_{max}$$

until r new tree nodes have been determined. The index q_{max} is the rth index in this sequence. In words, we are spawning generators on the nodes of the numerically smallest r-node subtree below, and including, node q. Having been assigned node numbers on the tree, the r new generators are initialized by assigning each new node with a primitive element in our enumeration. Finally, the child pointer, $c(\cdot)$, for the original generator and each of the new generators is replaced by successive doublings, until the new values are greater than q_{max}; in this way any new child generators spawned will be different from all previously created generators.

A small example will help to illustrate the spawning process just described. Suppose that 5 generators are needed at the start of a job. These would be placed at nodes 0,1,2,3, and 4 of the tree. Their child pointers are initially set as

$$c(0) = 8, \quad c(1) = 6, \quad c(2) = 5, \quad c(3) = 7, \quad c(4) = 9.$$

Now assume that, sometime later, generator 0 spawns 4 more children and that generator 3 spawns 6 more. The new children spawned by generator 0 will have tree node numbers 8, 16, 17, and 32, and we would have

$$c(8) = 34, \quad c(16) = 33, \quad c(17) = 35, \quad c(32) = 65.$$

In addition, the value $c(0)$ would be updated to 64. The new children spawned by generator 3 will have tree node numbers 7, 14, 15, 28, 29, and 30, with child pointer values

$$c(7) = 60, \quad c(14) = 58, \quad c(15) = 31, \quad c(28) = 57, \quad c(29) = 59, \quad c(30) = 61.$$

The value of $c(3)$ would then be updated to 56.

In [19] we implemented a library for parallel pseudorandom number generation based on a parameterization of additive lagged-Fibonacci recursions. There the parameterization was achieved through the seed, and the parallelization was accomplished via the same mapping of the parallel pseudorandom number generators onto the binary tree. It is our desire to model the software design for the new parallel LCG pseudorandom number generation package on this other parallel additive lagged-Fibonacci package. We have, in fact, implemented the same library routines for our LCG package with the algorithms discussed in this paper using the subroutine definitions given in [19].[4] At present the various Mersenne moduli we have specifically implemented are $m = 2^p - 1$ for $p = 31$, 61, 127, 521, and 607.

7. Conclusions and Future Directions.

In this paper we have described our approach to parallelizing Mersenne prime modulus LCGs via the parameterization of the multiplier. We have presented an efficient algorithm for the computation of the kth number relatively prime to a given, factored integer and have discussed the use of this algorithm in an implementation of a package for parallel LCGs. This package is designed using the same mapping onto the binary tree as a previous package for parallel pseudorandom number generation based on additive lagged-Fibonacci sequences.

The reader might be tempted to ask why two such similar packages for parallel pseudorandom number generation are needed by the Monte Carlo community. The scientific answer is that with more than one type of generator one can perform the same parallel calculation with each generator and compare the results statistically. If the mean and variance of both computations are similar, then one can be reassured that this particular computation is insensitive to the type of pseudorandom numbers used. It is often the case that subtle correlations in pseudorandom number generators can cause sensitive calculations to fail. More than one generator is essential to empirically rule out this problem in totally new calculations. Another reason why a parallel LCG package is desirable is less scientific than sociological. Many computational scientists that perform Monte Carlo computations have a distinct preference for their own pseudorandom number generator. This "native" generator

[4] The subroutines in the LCG library are identical to that in the lagged-Fibonacci sequence except that different data structures are needed in each case. Because of this identity, we refer the reader to the appendix of [19] for a description of the library. These packages are both part of the `SPRNG` package available at `www.ncsa.uiuc.edu/Apps/CMP/RNG`.

is often an LCG. Thus it is important to offer a parallel LCG to accommodate these preferences.

Another question that the reader may ask is why we have chosen to implement such a wide range of moduli, up to $2^{607}-1$! The period of such an LCG is $2^{607}-2$, which seems enormous, even for the fastest and most massively parallel system. Recall that not only is the period of the LCG a function of the modulus, but so is the total number of full-period LCGs. The period of a full-period LCG with prime modulus m is $O(m)$, while the the total number of full-period LCGs is $O(m/\log_2\log_2(m))$. Since many branching Monte Carlo computations require many available generators when using the binary tree mapping, we require large moduli in these situations to give us binary trees. One drawback of this fact is that we are forced to consider large moduli for reasons other than the total number pseudorandom numbers needed in a particular computation. This is a clear weakness for parallel LCGs since the computational cost per pseudorandom number of $O(\log_2(m))$ binary operations means that this cost scales up for with more processes.

With moduli on the order of hundreds of bits, it is reasonable to ask if the reduced cost of modular multiplication obtained when using a Mersenne prime is balanced by the increased cost required in computing $\pi_{m-1}^{-1}(k)$ during initialization. Obviously if the number of pseudorandom numbers required per process is large, this is an acceptable trade off of a stiff initialization cost for a reduced cost per pseudorandom number. However, in highly branched Monte Carlo computations one often uses only a few hundred to a few thousand pseudorandom numbers before branching. Thus one should consider other schemes that have a different balance between the cost per pseudorandom number and the initialization cost. Two possible approaches that are possible future research topics are to consider using Sophie-Germain primes instead of Mersenne primes as moduli and to consider splitting the sequences several times before computing a new multiplier.

Recall that a Sophie-Germain prime is a prime of the form $m=2q+1$, where q itself is prime. In this case $m-1=2q$, so the integers modulo $m-1$ that are relatively prime to $m-1$ are all the odds except q. In this very special case we get an explicit enumeration of the primitive elements modulo m. The price we must pay for this explicit enumeration is having to use standard modular multiplication. In practice, when the m approaches a few hundred bits in size, the cost of the shift and add modular reduction for a Mersenne prime is comparable to standard modular reduction. Thus it makes sense to consider using Sophie-Germain primes when large moduli are needed.

The second possible improvement is to increase the number of parallel processes available by using several subsequences from each full period cycle. This improvement would allow the same number of parallel processes to be furnished with a smaller modulus, and thus it would also speed up the cost of computing individual LCG pseudorandom numbers. One drawback to this approach is that very little research into the quality consequences of splitting full-period cycles for parallel pseudorandom generation has been done, [2, 12, 5, 6]. Much less research has been done into these results when both splitting and parameterization are used together.

Acknowledgments

This work is supported by DARPA/ITO under order number D343 held at The University of Southern Mississippi and the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign. The author would also like to thank Adam Meyerson of the Massachusetts Institute of Technology and Christopher Davis of the Pennsylvania State University for their careful implementations of these algorithms for our parallel linear congruential generation package.

References

1. S. Arno and K. Iobst, personal communication, (1991).
2. K. O. Bowman and M. T. Robinson, *Studies of random number generators for parallel processing*, Proceedings Second Conference: Hypercube Multiprocessors (Michael Heath, ed.), SIAM, Philadelphia, Pennsylvania, 1987, pp. 445-453.
3. J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman and S. S. Wagstaff, Jr., Parallel Comput. (1988), American Mathematical Society, Providence, Rhode Island.
4. M. Deleglise and J. Rivat, *Computing $\pi(x)$: the Meissel, Lehmer, Lagarias, Miller, Odlyzko method*, in the press, Math. Comput. (1995).
5. A. De Matteis and S. Pagnutti, *Parallelization of random number generators and long-range correlations*, Parallel Comput. **15** (1990), 155–164.
6. A. De Matteis and S. Pagnutti, *Long-range correlations in linear and non-linear random number generators*, Parallel Comput. **14** (1990), 207–210.
7. P. Frederickson, R. Hiromoto, T. L. Jordan, B. Smith and T. Warnock, *Pseudo-random trees in Monte Carlo*, Parallel Comput. **1** (1984), 175–180.
8. S. W. Golomb, *Shift Register Sequences*, Revised Edition, Aegean Park Press, Laguna Hills, California, 1982.
9. D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Second edition*, Addison-Wesley, Reading, Massachusetts, 1981.
10. L. Kuipers and H. Niederreiter, *Uniform distribution of sequences*, John Wiley and Sons, New York, 1974.
11. J. C. Lagarias, V. S. Miller and A. M. Odlyzko, *Computing $\pi(x)$: The Meissel-Lehmer method*, Math. Comput. **55** (1985), 537–560.
12. P. L'Ecuyer and S. Côté, *Implementing a random number package with splitting facilities*, ACM Trans. on Math. Soft. **17** (1991), 98–111.
13. D. H. Lehmer, *Mathematical methods in large-scale computing units*, Proc. 2nd Symposium on LargeScale Digital Calculating Machinery, Cambridge, Massachusetts, 1949, Harvard University Press, Cambridge, Massachusetts, 1949, pp. 141-146.
14. R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*, Cambridge University Press, Cambridge, London, New York, 1986.
15. M. Mascagni, S. A. Cuccaro, D. V. Pryor and M. L. Robinson, *A fast, high-quality, and reproducible lagged-Fibonacci pseudorandom number generator*, Comput. Physics **19** (1995), 211–219.
16. H. Niederreiter, *Quasi-Monte Carlo methods and pseudo-random numbers*, Bull. Amer. Math. Soc. **84** (1978), 957–1041.
17. H. Niederreiter, *Random number generation and quasi-Monte Carlo methods*, SIAM, Philadelphia, Pennsylvania, 1992.
18. O. E. Percus and M. H. Kalos, *Random number generators for MIMD parallel processors*, J. of Par. Distr. Comput. **6** (1989), 477–497.
19. D. V. Pryor, S. A. Cuccaro, M. Mascagni and M. L. Robinson, *Implementation and usage of a portable and reproducible parallel pseudorandom number generator*, in Proceedings of Supercomputing '94, IEEE, 1994, pp. 311–319.
20. W. Schmidt, *Equations over Finite Fields: An Elementary Approach*, Lecture Notes in Mathematics #536, Springer-Verlag, Berlin, Heidelberg, New York, 1976.

21. J. Spanier and E. M. Gelbard, *Monte Carlo Principles and Neutron Transport Problems*, Addison-Wesley, Reading, Massachusetts, 1969.

Box 10057 Southern Station; University of Southern Mississippi Hattiesburg, Mississippi 39406-0057 **USA**

E-mail address: Michael.Mascagni@usm.edu, URL: www.ncsa.uiuc.edu/Apps/CMP/RNG/mascagni