SOME METHODS OF PARALLEL PSEUDORANDOM NUMBER GENERATION

MICHAEL MASCAGNI*

Abstract. We detail several methods used in the production of pseudorandom numbers for scalable systems. We will focus on methods based on parameterization, meaning that we will not consider splitting methods. We describe parameterized versions of the following pseudorandom number generation:

1. linear congruential generators
2. linear matrix generators
3. shift-register generators
4. lagged-Fibonacci generators
5. inversive congruential generators

We briefly describe the methods, detail some advantages and disadvantages of each method and recount results from number theory that impact our understanding of their quality in parallel applications. Several of these methods are currently part of scalable library for pseudorandom number generation, called the `SPRNG` package available at the URL: `www.ncsa.uiuc.edu/Apps/CMP/RNG`.

Key words. pseudorandom number generation, parallel computing, linear congruential, lagged-Fibonacci, inversive congruential, shift-register

AMS(MOS) subject classifications. 65C10, 65Y05, 68Q22

1. Introduction.

Monte Carlo applications are widely perceived as embarrassingly parallel. The truth of this notion depends, to a large extent, on the quality of the parallel random number generator used. It is widely assumed that with N processors executing N copies of a Monte Carlo calculation, the pooled result will achieve a variance N times smaller than a single instance of this calculation in the same amount of time. This is true only if the results in each processor are statistically independent. In turn, this will be true only if the streams of random numbers generated in each processor are independent. This paper will briefly present several methods for parallel pseudorandom number generation and discuss pros and cons for each method. If the reader is interested in background material on pseudorandom number generation in general, consult the following references: [13], [16], [34], [36].

Here we are interested, exclusively, with methods for obtaining **p**arallel **p**seudorandom **n**umber **g**enerator**s** (PPRNGs) via parameterization. The exact meaning of parameterization depends on the type of PRNG under discussion, but we wish to distinguish parameterization from splitting methods. We will not be considering producing parallel streams of pseudorandom numbers by taking substreams from a single, long-period PRNG. For

* Program in Scientific Computing **and** Department of Mathematics, Southern Station, Box 10057, University of Southern Mississippi, Hattiesburg, MS 39406-0057, e-mail: `Michael.Mascagni@usm.edu`, web: `www.ncsa.uiuc.edu/Apps/CMP/RNG/mascagni`. This work was partially supported by DARPA under contract DABT-63-95-C-0123.

readers interested in splitting methods and the consequences of using split streams in parallel please consult: [5], [7], [8], [9], [11], [17]. In general, we seek to determine a parameter in the underlying recursion of the PRNG that can be varied. Each valid value of this parameter will lead to a recursion that produces a unique, full-period stream of pseudorandom numbers. We then discuss efficient means to specify valid parameter values and consider these choices in terms of the quality of the pseudorandom numbers produced.

The plan of the paper is as follows. In §2 we present two methods for parameterizing **l**inear **c**ongruential **g**enerator**s** (LCGs). In §3 we briefly present and discuss linear matrix methods for parallel pseudorandom number generation. In §4 we present a parameterization of another linear method: **s**hift-**r**egister **g**enerator**s** (SRGs). This parameterization is analogous to one of the LCG parameterizations presented in §2. In §5 we consider the parallel parameterization of so-called lagged-Fibonacci generators, and in §6 we present a parallel nonlinear random number generator called the **i**nversive **c**ongruential **g**enerator (ICG). Finally in §7 we discuss open problems, alert the reader to available packages based on some of the algorithms discussed, and provide concluding remarks.

2. Linear Congruential Generators. The most commonly used generator for pseudorandom numbers is the **l**inear **c**ongruential **g**enerator (LCG). The LCG was first proposed for use by Lehmer, [18], and is referred to as the Lehmer generator in the early literature. The linear recursion underlying LCGs is:

$$x_n = ax_{n-1} + b \pmod{m}. \tag{2.1}$$

The x_n's are integer residues modulo m, and a uniform pseudorandom number in [0,1] is produced via $z_n = x_n/m$. The constants of the LCG in 2.1 are referred to as the modulus, m, the multiplier, a, and the additive constant, c. The initial value of the LCG, x_0, is often called the seed.

The most important parameter of an LCG is the modulus, m. Its size constrains the period, and for implementational reasons it is always chosen to be either prime or a power-of-two. Based on which type of modulus is chosen, there is a different parameterization method. When m is prime, a method based on using the multiplier, a, as the parameter has been proposed. The rationale for this choice is outlined in, [27], and leads to several interesting computational problems.

2.1. Prime Modulus. Given we wish to parameterize a when m is prime we must determine first the family of permissible a's. A condition on a when m is prime to obtain the maximal period (of length $m-1$ in this case) is that a must be a primitive element modulo m, [13].[1] Given

[1] An integer, a, is primitive modulo m if the set of integers $\{a^i \pmod{m} | 1 \leq i \leq m-1\}$ equals the set $\{1 \leq i \leq m-1\}$.

primitivity, one can use the following fact: if a and α are primitive elements modulo m then $\alpha = a^i \pmod{m}$ for some i relatively prime to $\phi(m)$. Note that when m is prime that $\phi(m) = m-1$. Thus a single, reference, primitive element, a, and an explicit enumeration of the integers relatively prime to $m-1$ furnish an explicit parameterization for the jth primitive element, a_j as $a_j = a^{\ell_j} \pmod{m}$ where ℓ_j is the jth integer relatively prime to $m-1$. Given an explicit factorization of $m-1$, [3], efficient algorithms for computing ℓ_j can be found in a recent work of the author, [27]. An interesting open question in this regard is whether the overall efficiency of this PPRNG is minimized by choosing the prime modulus to minimize the cost of computing ℓ_j or to minimize the cost of modular multiplication modulo m.

Given this scheme there are some positive and negative features to be mentioned. A motivation for this scheme is that a common theoretical measure of the correlation among parallel streams predicts little correlation. This measure is based on 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:

$$C(k) = \sum_{n=0}^{k-1} e^{\frac{2\pi i}{m} x_n}. \tag{2.2}$$

If the x_n are periodic and k is the period, then 2.2 is called a full-period exponential sum. If x_n is periodic and k is less than the full period, then 2.2 is a partial-period exponential sum. Examining 2.2 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 $|C(k)| = O(\sqrt{k})$, [14]. 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:

$$C(i,j,k) = \sum_{n=0}^{k-1} e^{\frac{2\pi i}{m}(x_{i+n} - y_{j+n})}. \tag{2.3}$$

Here the sum has k terms and begin 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, [38]. We will take the same approach here. Suppose we have j full-period LCGs defined by $x_{k_n} = a^{\ell_i} x_{k_{n-1}} \pmod{m}$, $0 \leq k < j$. All of the pairwise full-period exponential sum cross-correlations are known to satisfy, [39]:

$$|C(m)| \leq \left(\left[\max_k \ell_k \right] - 1 \right) \sqrt{m}. \tag{2.4}$$

The choice of the exponents, ℓ_k, that minimizes 2.4 is that ℓ_j is made the jth integer relatively prime to $m-1$. This necessitates an algorithms to compute this jth integer relatively prime to an integer with known factorization, $m-1$. This is discussed at great length in [27]; however, two important open questions remain: (1) is it more efficient overall to choose m to be amenable to fast modular multiplication or fast calculation of the jth integer relatively prime to $m-1$, and (2) does the good interstream correlation of 2.4 also ensure good intrastream independence via the spectral test? The first of these questions is of practical interest to performance, the second; however, if answered negatively, makes such techniques less attractive for parallel pseudorandom number generation.

2.2. Power-of-two Modulus. An alternative way to use LCGs to make a PPRNG is to parameterize the additive constant in equation 2.1 when the modulus is a power-of-two, i.e., to $m = 2^k$ for some integer $k > 1$. This is a technique first proposed by Percus and Kalos, [37], to provide a PPRNG for the NYU Ultracomputer. It has some interesting advantages over parameterizing the multiplier; however, there are some considerable disadvantages in using power-of-two modulus LCGs.

The parameterization chooses a set of additive constants $\{b_j\}$ that are pairwise relatively prime, i.e. $\gcd(b_i, b_j) = 1$ when $i \neq j$. A prudent choice is to let b_j be the jth prime. This both ensures the pairwise relative primality and is the largest set of such residues. With this choice certain favorable interstream properties can be theoretically derived from the spectral test, [37]. However, this choice necessitates a method for the difficult problem of computing the jth prime. In their paper, Percus and Kalos do not discuss this aspect of their generator in detail, partly due to the fact that they expect to provide only a small number of PRNGs. When a large number of PPRNGs are to be provided with this method, one can use fast algorithms for the computation of $\pi(x)$, the number of primes less than x, [6], [15] . This is the inverse of the function which is desired, so we designate $\pi^{-1}(j)$ as the jth prime. The details of such an implementation need to be specified, but a very related computation for computing the jth integer relatively prime to a given set of integers is given in, [27]. It is believed that the issues for computing $\pi^{-1}(j)$ are similar.

One important advantage of this parameterization is that there is an interstream correlation measure based on the spectral test that suggests that there will be good interstream independence. Given that the spectral test for LCGs essentially measures the quality of the multiplier, this sort of result is to be expected. A disadvantage of this parameterization is that to provide a large number of streams, computing $\pi^{-1}(j)$ will be necessary. Regardless of the efficiency of implementation, this is known to be a difficult computation with regards to its computational complexity. Finally, one of the biggest disadvantages to using a power-of-two modulus is the fact the least significant bits of the integers produced by these LCGs have extremely

short periods. If $\{x_n\}$ are the residues of the LCG modulo 2^k, with properly chosen parameters, $\{x_n\}$ will have period 2^k. However, $\{x_n \pmod{2^j}\}$ will have period 2^j for all integers $0 < j < k$, [13]. In particular, this means the the least-significant bit of the LCG with alternate between 0 and 1. This is such a major short coming, that it motivated us to consider parameterizations of prime modulus LCGs as discussed in §2.1.

3. Linear Matrix Generators. Recent trends in computer architecture have motivated researchers to study methods of generating pseudorandom vectors, [32], [33]. These techniques are appropriate to vector architectures, but are not well suited to parallel machines due to the algorithm's lack of data locality. **L**inear **m**atrix **g**enerator**s** (LMGs) are given by the following equation:

$$\mathbf{x}_n = \mathbf{A}\mathbf{x}_{n-1} \pmod{m}. \tag{3.1}$$

Here the matrix $\mathbf{A}$ is $k \times k$ and the vector $\mathbf{x}_n$ is k-dimensional. One obtains a uniform pseudorandom vector by forming $\mathbf{z}_n = \mathbf{x}_n/m$. When m is prime, the maximal period for the LMG is $m^k - 1$ if and only if $\mathbf{A}$ has a characteristic polynomial that is primitive modulo m, [34]. In this situation the k-tuples produced by equation 3.1 will pass the k-dimensional equidistribution test as well as an LCG passed the one-dimensional equidistribution test. This is to be expected as the full-period of the LMG produces all possible k-tuples modulo m. For dimensions larger than k, behavior analogous to the LCG in two or more dimensions is seen.

4. Shift-Register Generators. **S**hift **r**egister **g**enerator**s** (SRGs) are linear recursions modulo 2, [12], [19], [40], of the form:

$$x_{n+k} = \sum_{i=0}^{k-1} k a_i x_{n+i} \pmod{2}, \tag{4.1}$$

where the a_i's are either 0 or 1. An alternative way to describe this recursion is to specify the kth degree binary characteristic polynomial, [20]:

$$f(x) = x^k + \sum_{i=0}^{k-1} a_i x^i \pmod{2}. \tag{4.2}$$

To obtain the maximal period of $2^k - 1$, a sufficient condition is that $f(x)$ be a primitive kth degree polynomial modulo 2. If only a few of the a_i's are 1, then 4.1 is very cheap to evaluate. Thus people often use known primitive trinomials to specify SRG recursions. This leads to very efficient, two-term, recursions.

There are two ways to make pseudorandom integers out of the bits produced by 4.1. The first, called the digital multistep method, takes successive bits from 4.1 to form an integer of desired length. Thus, with the

digital multistep method, it requires n iterations of 4.1 to produce a new n-bit pseudorandom integer. The second method, called the generalized feedback shift-register, creates a new n-bit pseudorandom integer for every iteration of 4.1. This is done by constructing the n-bit word from x_{n+k} and $n-1$ other bits from the k bits of SRG state. While these two methods seem different, they are very related, and theoretical results for one always hold for the other. Reader's interested in more general information on SRGs should consult the references: [12], [19], [40]. One way to parameterize SRGs is analogous to the LCG parameterization discussed in §2.1. There we took the object that made the LCG full-period, the primitive root multiplier, and found a representation for all of them. Using this analogy we identify the primitive polynomial in the SRG as the object to parameterize. We begin with a known primitive polynomial of degree k, $p(x)$. It is known that only certain decimations of the output of a maximal-period shift register are themselves maximal and unique with respect to cyclic reordering, [20]. We seek to identify those. The number of decimations that are both maximal-period and unique when $p(x)$ is primitive modulo 2 and k is a Mersenne exponent is $\frac{2^k-2}{k}$. If a is a primitive root modulo the prime 2^k-1, then the residues $a^i \pmod{2^k-1}$ for $i=1$ to $\frac{2^k-2}{k}$ form a set of all the unique, maximal-period decimations. Thus we have a parameterization of the maximal-period sequences of length 2^k-1 arising from primitive degree k binary polynomials through decimations.

The entire parameterization goes as follows. Assume the kth stream is required, compute $d_k \equiv a^k \pmod{2^k-1}$ and take the d_kth decimation of the reference sequence produced by the reference primitive polynomial, $p(x)$. This can be done quickly with polynomial algebra. Given a decimation of length $2k+1$, this can be used as input the Berlekamp-Massey algorithm to recover the primitive polynomial corresponding to this decimation. The Berlekamp-Massey algorithm finds the minimal polynomial that generates a given sequence, [30] in time linear in k.

This parameterization is relatively efficient when the binary polynomial algebra is implemented correctly. However, there is one major drawback to using such a parameterization. While the reference primitive polynomial, $p(x)$, may be sparse, the new polynomials need not be. By a sparse polynomial we mean that most of the a_i's in 4.1 are zero. The cost of stepping 4.1 once is proportional to the number of non-zero a_i's in 4.1. Thus we can significantly increase the bit-operational complexity of a SRG in this manner.

The fact that the parameterization methods for prime modulus LCGs and SRGs is no accident. Both are based on maximal period linear recursions over a finite field. Thus the discrepancy and exponential sum results for both the types of generators are similar, [34]. However, a result for SRGs analogous to that in 2.4 is not known. It is open whether or not such a cross-correlation result holds for SRGs, but it is widely thought to.

5. Lagged-Fibonacci Generators. In the previous sections we have discussed generators that can be parallelized by varying a parameter in the underlying recursion. In this section we discuss the **a**dditive **l**agged-**F**ibonacci **g**enerator (ALFG): a generator that can be parameterized through its initial values. The ALFG can be written as:

$$x_n = x_{n-j} + x_{n-k} \pmod{2^m}, \quad j < k. \tag{5.1}$$

In recent years the ALFG has become a popular generator for serial as well as scalable parallel machines, [21]. In fact, the generator with $j = 5$, $k = 17$, and $m = 32$ was the standard PPRNG in Thinking Machines Connection Machine Scientific Subroutine Library. This generator has become popular for a variety of reasons: (1) it is easy to implement, (2) it is cheap to compute using 5.1, and (3) the ALFG does well on standard statistical tests, [24].

An important property of the ALFG is that the maximal period is $(2^k - 1)2^{m-1}$. This occurs for very specific circumstances, [2], [25], from which one can infer that this generator has $2^{(k-1)\times(m-1)}$ different full-period cycles, [29]. This means that the state space of the ALFG is toroidal, with equation 5.1 providing the algorithm for movement in one of the torus dimension. It is clear that finding the algorithm for movement in the other dimension is the basis of a very interesting parameterization. Since 5.1 cycles over the full period of the ALFG, one must find a seed that is not in a given full-period cycle to move in the second dimension. The key to moving in this second dimension is to find an algorithm for computing seeds in any given full-period cycle.

A very elegant algorithm for movement in this second dimension is based on a simple enumeration as follows. One can prove that the initial seed, $\{x_0, x_1, \ldots, x_{k-1}\}$, can be bit-wise initialized using the following template:

(5.2)

m.s.b				l.s.b.	
b_{m-1}	b_{m-2}	...	b_1	b_0	
□	□	...	0	0	x_{k-1}
0	□	...	□	0	x_{k-2}
⋮	⋮	⋮	⋮	⋮	
□	0	...	□	0	x_1
□	□	...	□	1	x_0

Here each square is a bit location to be assigned. Each unique assignment gives a seed in a provably distinct full-period cycle, [29]. Note that here the least-significant bits, b_0 are specified to be a fixed, non-zero, pattern. If one allows an $O(k^2)$ precomputation to find a particular least-significant-bit

pattern then the template is particularly simple:

(5.3)

m.s.b				l.s.b.	
b_{m-1}	b_{m-2}	...	b_1	b_0	
□	□	...	□	$b_{0\,k-1}$	x_{k-1}
□	□	...	□	$b_{0\,k-2}$	x_{k-2}
⋮	⋮	⋮	⋮	⋮	
□	□	...	□	b_{01}	x_1
0	0	...	0	1	x_0

Given the elegance of this explicit parameterization, one may ask about the exponential sum correlations between these parameterized sequences. It is known that certain sequences are more correlated than others as a function of the similarity in the least-significant bits in the template for parameterization, [26]. However, it is easy to avoid all but the most uncorrelated pairs in a computation, [38]. In this case there is extensive empirical evidence that the full-period exponential sum correlation between streams is $O(\sqrt{(2^k-1)2^{m-1}})$, the square root of the full-period. This is essentially optimal. Unfortunately, there is no analytic proof of this result, and improvement of the best known analytic result, [26], is an important open problem in the theory of ALFGs.

Another advantage of the ALFG is that one can implement these generators directly with floating-point numbers to avoid the constant conversion from integer to floating-point that accompanies the use of other generators. This is a distinct speed improvement when only floating-point numbers are required in the Monte Carlo computation. However, care must be taken to maintain the identity of the corresponding integer recursion when using the floating-point ALFG in parallel to maintain the uniqueness of the parallel streams. A discussion of how to ensure fidelity with the integer streams can be found in [1].

An interesting cousin of the ALFG is the **m**ultiplicative **l**agged-**F**ibonacci **g**enerator (MLFG). It is defined by:

$$x_n = x_{n-j} \times x_{n-k} \pmod{2^m}, \quad j < k. \tag{5.4}$$

While this generator has a maximal-period of $(2^k-1)2^{m-3}$, which is a quarter the length of the corresponding ALFG, [25], it has empirical properties considered to be superior to ALFGs, [24]. Of interest for parallel computing is that a parameterization analogous to that of the ALFG exists for the MLFG, [28].

6. Inversive Congruential Generators. An important new type of PRNG that, as yet, has not found any widely distributed implementation is the **i**nversive **c**ongruential **g**enerator (ICG). This generator comes in two versions, the recursive ICG, [10], [31], and the explicit ICG, [35] . The formula for the recursive ICG is:

$$x_n = a\overline{x}_{n-1} + b \pmod{m}, \tag{6.1}$$

while the explicit ICG has formula:

$$x_n = \overline{an + b} \pmod{m}. \tag{6.2}$$

In both the above equations $\bar{c}$ denotes the multiplicative inverse modulo m in the sense that $c\bar{c} \equiv 1 \pmod{m}$ when $c \neq 0$, and $\bar{0} = 0$.

An advantage of ICGs over LCGs are that tuples made from ICGs do not fall in hyperplanes, [22], [23]. The quantification of this is the lattice test. We say that a generator passes the k-dimensional lattice test if vectors made up of k-tuples from the generator span the k-dimensional vector space taken modulo m. ICGs have optimal behavior for the lattice test in that they pass for $k \leq m - 1$. Another advantage of ICGs is that they are not nearly as uniformly distributed over their full period as generators from linear recursions. They behave more like truly random numbers, [34].

An interesting fact about the (non)lattice structure of tuples from explicit ICGs has ramifications for PPRNGs via parameterization. Consider the parameterized explicit ICG: $x_{k_n} = \overline{a_k n + b_k} \pmod{m}$, $0 \leq k \leq s$. If the residues modulo m, $a_0\overline{b_0}, \ldots, a_{s-1}\overline{b_{s-1}}$, are all distinct, then set of all the s-tuples $(x_{0_j}, x_{1_j}, \ldots, x_{s-1_j})$, $j = 0$ to $m - 1$, appear to be extremely well distributed as follows. Take any s-dimensional hyperplane passing through the origin, it will intersect at most $s - 1$ of these points.

While ICGs have some very compelling equidistribution properties, they remain out of the mainstream random number packages. This is due to both the fact that ICGs are relatively unknown outside of the mathematical random number generation community and that the cost of doing modular inversion is quite considerable. If we consider the cost of modular multiplication to be the cost unit, then modular inversion is $O(\log_2 m)$. For certain applications, this extra cost may be worth it, but in most applications that involve parallel computers computational efficiency is an important factor. Thus the author expects to see ICGs available in some serial random number packages; however, he doubts if ICGs will be implemented for parallel machines soon.

7. Conclusions and Open Problems. The parallelization of the ALFG is the basis for the default generator in the **S**calable **P**arallel **R**andom **N**umber **G**eneration (SPRNG) library available from URL: `www.ncsa.uiuc.edu/Apps/CMP/RNG`. In addition, the SPRNG library includes (or will soon include) the two parameterized LCGs described above and the parameterized SRG. In the SPRNG library the same technique is used to implement the parallelization via parameterization using a mapping of the generators (as indexed by parameter) onto the binary tree. This is a convenient canonical mapping that provides each generator with a subtree of successors that are disjoint from subtrees rooted at other generators. This allows the operation of this PPRNG in a MIMD execution environment with PRNG spawning. In application, such as neutronics, one often needs to dynamically generate new generators. The disjoint subtrees of

processors allows generators to be assigned uniquely and reproducibly.

While care has been taken in constructing generators for the SPRNG package, the designers realize that there is no such thing as a PRNG that behaves flawless for every application. This is even more true when one considers using scalable platforms for Monte Carlo. The underlying recursions that are used are for PRNGs are simple, and so they inevitably have regular structure. This deterministic regularity permits analysis of the sequences and is the PRNG's Achilles heel. Thus any large Monte Carlo calculation must be viewed with suspicion as an unfortunate interplay between the application and PRNG may result in spurious results. The only way to prevent this is to treat each new Monte Carlo derived result as an experiment that must be controlled. The tools required to control problems with the PRNG include the ablility to use another PRNG in the same calculation. In addition, one must be able to use new PRNGs as well. These capabilities as well as parallel and serial tests of randomness, [4], are components that make the SPRNG package unique among tools for parallel Monte Carlo.

REFERENCES

[1] R. P. Brent, *Uniform Random Number Generators for Supercomputers* in *Proceedings Fifth Australian Supercomputer Conference*, SASC Organizing Committee, pp. 95–104, 1992.

[2] R. P. Brent, *On the periods of generalized Fibonacci recurrences*, Mathematics of Computation, 1994, **63**: 389–401.

[3] J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman and S. S. Wagstaff, Jr., *Factorizations of $b^n \pm 1$ $b = 2, 3, 5, 7, 10, 11, 12$ up to high powers*, *Contemporary Mathematics* Volume 22, Second Edition, American Mathematical Society, Providence, Rhode Island, 1988.

[4] S. A. Cuccaro, M. Mascagni and D. V. Pryor, *Techniques for testing the quality of parallel pseudorandom number generators*, in *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, Philadelphia, Pennsylvania, pp. 279–284, 1995.

[5] I. Deák, *Uniform random number generators for parallel computers*, Parallel Computing, 1990, **15**: 155–164.

[6] M. Deleglise and J. Rivat, *Computing $\pi(x)$: the Meissel, Lehmer, Lagarias, Miller, Odlyzko method*, Mathematics of Computation, 1996, **65**: 235–245.

[7] A. De Matteis and S. Pagnutti, *Parallelization of random number generators and long-range correlations*, Parallel Computing, 1990, **15**: 155–164.

[8] A. De Matteis and S. Pagnutti, *A class of parallel random number generators*, Parallel Computing, 1990, **13**: 193–198.

[9] A. De Matteis and S. Pagnutti, *Long-range correlations in linear and non-linear random number generators*, Parallel Computing, 1990, **14**: 207–210.

[10] J. Eichenauer and J. Lehn, *A nonlinear congruential pseudorandom number generator*, Statist. Hefte, 1986, **37**: 315–326.

[11] P. Frederickson, R. Hiromoto, T. L. Jordan, B. Smith and T. Warnock, *Pseudo-random trees in Monte Carlo*, Parallel Computing, 1984, **1**: 175–180.

[12] S. W. Golomb, *Shift Register Sequences*, Revised Edition, Aegean Park Press, Laguna Hills, California, 1982.

[13] D. E. Knuth, *Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Second edition, Addison-Wesley, Reading, Massachusetts, 1981.

[14] L. KUIPERS AND H. NIEDERREITER, *Uniform distribution of sequences*, John Wiley and Sons: New York, 1974.

[15] J. C. LAGARIAS, V. S. MILLER AND A. M. ODLYZKO, *Computing $\pi(x)$: The Meissel-Lehmer method*, Mathematics of Computation, 1985, **55**: 537–560.

[16] P. L'ECUYER, *Random numbers for simulation*, Communications of the ACM, 1990, **33**: 85–97.

[17] P. L'ECUYER AND S. CÔTÉ, *Implementing a random number package with splitting facilities*, ACM Trans. on Mathematical Software, 1991, **17**: 98–111.

[18] D. H. LEHMER, *Mathematical methods in large-scale computing units*, in *Proc. 2nd Symposium on LargeScale Digital Calculating Machinery*, Harvard University Press: Cambridge, Massachusetts, 1949, pp. 141–146.

[19] T. G. LEWIS AND W. H. PAYNE, *Generalized feedback shift register pseudorandom number algorithms*, Journal of the ACM, 1973, **20**: 456–468.

[20] R. LIDL AND H. NIEDERREITER, *Introduction to finite fields and their applications*, Cambridge University Press: Cambridge, London, New York, 1986.

[21] J. MAKINO, *Lagged-Fibonacci random number generator on parallel computers*, Parallel Computing, 1994, **20**: 1357–1367.

[22] G. MARSAGLIA, *Random numbers fall mainly in the planes*, Proc. Nat. Acad. Sci. U.S.A., 1968, **62**: 25–28.

[23] G. MARSAGLIA, *The structure of linear congruential sequences*, in *Applications of Number Theory to Numerical Analysis*, S. K. Zaremba, Ed., Academic Press, New York, 1972, pp. 249–285.

[24] G. MARSAGLIA, *A current view of random number generators*, in *Computing Science and Statistics: Proceedings of the XVIth Symposium on the Interface*, 1985, pp. 3–10.

[25] G. MARSAGLIA AND L.-H. TSAY, *Matrices and the structure of random number sequences*, Linear Alg. and Applic., 1985 , **67**: 147–156.

[26] M. MASCAGNI, M. L. ROBINSON, D. V. PRYOR AND S. A. CUCCARO, *Parallel pseudorandom number generation using additive lagged-Fibonacci recursions*, Springer Verlag Lecture Notes in Statistics, 1995, **106**: 263–277.

[27] M. MASCAGNI, *Parallel linear congruential generators with prime moduli*, 1997, IMA Preprint #1470 and submitted.

[28] M. MASCAGNI, *A parallel non-linear Fibonacci pseudorandom number generator*, 1997, abstract, 45th SIAM Annual Meeting.

[29] M. MASCAGNI, S. A. CUCCARO, D. V. PRYOR AND M. L. ROBINSON, *A fast, high-quality, and reproducible lagged-Fibonacci pseudorandom number generator*, Journal of Computational Physics, 1995, **15**: 211–219.

[30] J. L. MASSEY, *Shift-register synthesis and BCH decoding*, IEEE Trans. Information Theory, 1969, **IT-15**: 122–127.

[31] H. NIEDERREITER, *Statistical independence of nonlinear congruential pseudorandom numbers*, Montash. Math., 1988, **106**: 149–159.

[32] H. NIEDERREITER, *Statistical independence properties of pseudorandom vectors produced by matrix generators*, J. Comput. and Appl. Math., 1990, **31**: 139–151.

[33] H. NIEDERREITER, *Recent trends in random number and random vector generation*, Ann. Operations Research, 1991, **31**: 323–346.

[34] H. NIEDERREITER, *Random number generation and quasi-Monte Carlo methods*, SIAM: Philadelphia, Pennsylvania, 1992.

[35] H. NIEDERREITER, *On a new class of pseudorandom numbers for simulation methods*, J. Comput. Appl. Math., 1994, **65**: 159–167.

[36] S. K. PARK AND K. W. MILLER, *Random number generators: good ones are hard to find*, Communications of the ACM, 1988, **31**: 1192–1201.

[37] O. E. PERCUS AND M. H. KALOS, *Random number generators for MIMD parallel processors*, J. of Par. Distr. Comput., 1989, **6**: 477–497.

[38] D. V. PRYOR, S. A. CUCCARO, M. MASCAGNI AND M. L. ROBINSON, IMPLEMENTATION AND USAGE OF A PORTABLE AND REPRODUCIBLE PARALLEL PSEUDORAN-

DOM NUMBER GENERATOR, in *Proceedings of Supercomputing '94*, IEEE, 1994, pp. 311–319.

[39] W. SCHMIDT, *Equations over Finite Fields: An Elementary Approach*, Lecture Notes in Mathematics #536, Springer-Verlag: Berlin, Heidelberg, New York, 1976.

[40] R. C. TAUSWORTHE, *Random numbers generated by linear recurrence modulo two*, Mathematics of Computation, 1965, **19**: 201–209.