Parallel Computing 35 (2009) 29–37

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Scalable parallel multiple recursive generators of large order

Lih-Yuan Deng [a,*], Huajiang Li [b], Jyh-Jen Horng Shiau [c]

[a] Department of Mathematical Sciences, The University of Memphis, Memphis, TN 38152, USA
[b] Quintiles Inc., Overland Park, KS 66211, USA
[c] Institute of Statistics, National Chiao Tung University, Hsinchu 30010, Taiwan, ROC

ARTICLE INFO

Article history:
Received 2 August 2005
Received in revised form 31 December 2007
Accepted 26 September 2008
Available online 10 October 2008

Keywords:
DX-*k*-*s* generator
Generalized Mersenne prime
Irreducible polynomial
Linear congruential generator
MRG
Primitive polynomial

ABSTRACT

To speed up the process of performing a large statistical simulation study, it is natural and common to divide the large-scale simulation task into several relatively independent sub-tasks in a way that these sub-tasks can be handled by individual processors in parallel. To obtain a good overall simulation result by synthesizing results from these sub-tasks, it is crucial that good parallel random number generators are used. Thus, designing suitable and independent uniform random number generators for the sub-tasks has become a very important issue in large-scale parallel simulations. Two commonly used uniform random number generators, linear congruential generator (LCG) and multiple recursive generator (MRG), have served as backbone generators for some parallel random number generators constructed in the past. We will discuss some general construction methods. A systematic leapfrog method to automatically choose different multipliers for LCGs to have the maximum-period and a method to construct many maximum-period MRGs from a single MRG are available in the literature. In this paper, we propose to combine both approaches to generate different MRGs "randomly", quickly and automatically, while retaining the maximum-period property.

Published by Elsevier B.V.

1. Introduction

Linear congruential generator (LCG) and multiple recursive generator (MRG) are two of the most popular uniform random number generators (RNGs). They follow a similar procedure: choose a starting seed or seeds, compute a linear recurrence equation with properly chosen multipliers and, if any, some additive constants, and compute the next number under a prime modulus.

For a large-scale simulation study, it is common to run simulations in parallel on several processors to speed up the simulation process. To perform a parallel simulation on different processors, we need a method of generating "independent" streams of random numbers. It is possible to achieve this by either choosing different (a) moduli, (b) additive constants, (c) starting seeds, or (d) multipliers for different processors. For more details on various methods, please see Deng et al. [5], Mascagni [18], Mascagni and Srinivasan [19], Srinivasan et al. [21], and L'Ecuyer et al. [15].

The approaches (a) and (b) are not recommended. Unless appropriate parameters have been pre-computed and stored, using different moduli is not feasible. In general, storing pre-computed parameters is not a good idea unless the number of processors is small. These methods are *not scalable* because they cannot easily be applied in a large-scale simulation where the number of available processors are unknown. Choosing different additive constants is also not advisable and is only suitable for LCG. This approach also introduces serious inter-stream correlation. Thus, it is more common to use either (c) or (d). Choosing different starting seeds for different processors is popular with some efficient skip-ahead schemes. Since LCG's

* Corresponding author.
E-mail address: lihdeng@memphis.edu (L.-Y. Deng).

0167-8191/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.parco.2008.09.012

period length is short, there is a problem of overlapping between two "independent" streams. As to MRGs, with the long period, one can avoid the overlapping problem by skipping very far ahead and computing/storing various sets of starting seeds. We remark that the jump-ahead scheme might result in power-of-two correlations between computations in different processors. However, this problem can be easily remedied by using a modulus that is not power of two.

Alternatively, the approach (d) that chooses different multipliers for different processors can avoid the inter-stream correlation problem. The main objective of this paper is to provide a method to produce quickly and automatically different multipliers, for which the corresponding generators are of the maximum-period. The term "maximum-period" is defined as the maximum period that a generator in a particular class can possibly achieve.

For LCG, Deng et al. [5] proposed a systematic leapfrog method to automatically choose different multipliers such that the corresponding LCGs are of the maximum-period. For MRG, Deng [2] proposed an efficient method to find a large order MRG with the maximum-period and a method to construct maximum-period MRGs from a single MRG. In this paper, we extend his approach to generate different MRGs "randomly", quickly and automatically, while retaining the maximum-period property.

In addition to speeding up the simulation process, parallel random number generators can be useful in some Markov Chain Monte Carlo (MCMC) methods. One of the simple and widely used MCMC methods is called Gibbs sampling introduced by Geman and Geman [9]. Instead of generating independent sequences as most of the Monte Carlo simulations do, Gibbs sampling generates dependent Markov chains that will converge to the desired stationary distribution. One of the difficult problems in MCMC is knowing when to stop the iteration. The answer depends on when the sequence is converging to a stable distribution. In practice, it is common to run several independent Gibbs sampling sequences simultaneously and watch their sampling behaviors to determine the convergence. These sequences (streams) can be run on a multi-processor computer system or on several different processes of a same computer.

In Section 2, we review some results of the classical LCG and MRG. A class of efficient MRGs called DX-k-s generators, proposed by Deng and Xu [7], to be used as the backbone generators for parallel RNGs in this paper is also reviewed. In Section 3, we briefly discuss some general approaches of parallel RNGs. In Section 4, we introduce an automatic generating method for parallel MRGs. In Section 5, we give tables of many maximum-period MRGs of very large orders ranging from 101 to 4001. In addition, an illustrative example of our proposed automatic generating method is given with $k = 4001$. We show that up to 1,071,535,582 different MRG generators, each with the maximum-period of $10^{37,333.5}$, can be systematically constructed. Finally, in Section 6, we report the latest results on the search of DX generators. For k ranging from 5003 to 10,007, several DX-k-s generators have been found and reported in Deng [4]. We can then apply the proposed method to any of these newly found generators to construct quickly many distinct MRGs, for which the largest maximum-period is as large as $10^{93,383.7}$.

2. LCG and MRG

Until recently, the LCG proposed by Lehmer [16] is the simplest and most widely used uniform random number generator

$$X_i = BX_{i-1} \bmod p, \quad i \geqslant 1, \quad U_i = X_i/p, \tag{1}$$

where the multiplier B and prime modulus p are positive integers, and X_0 is any non-zero seed. For a prime p, the maximum-period of an LCG in (1) is $p-1$, which is achieved if B is chosen to be a primitive root of a finite field with p elements, $\mathbb{Z}_p = \{0, 1, 2, \ldots, p-1\}$. An integer B is a primitive root modulo a prime number p, if and only if $B^{(p-1)/q} \neq 1 \bmod p$ for any prime factor q of $p-1$. Also, for a prime modulus p, B is a primitive root or a primitive element of p if and only if the above LCG achieves the maximum-period. Following this definition, when the modulus is a composite number m, B is called a primitive element modulo m if its corresponding LCG yields a maximum-period. Knuth [12] described a general method of finding such primitive elements, using the "Chinese remainder algorithm" for a system of congruential equations.

MRG is a natural extension of LCG, which generates the next number based on a linear combination of the most recent k numbers:

$$X_i = (\alpha_1 X_{i-1} + \cdots + \alpha_k X_{i-k}) \bmod p, \quad i \geqslant k, \tag{2}$$

where the multipliers $\alpha_1, \ldots, \alpha_k$ and prime modulus p are all positive integers, $\{X_0, \ldots, X_{k-1}\}$ are initial seeds which can be chosen arbitrarily from $\mathbb{Z}_p$ with the restriction that not all of them are equal to zero. Here, the integer k is called the order of MRG. When $k = 1$, MRG is degenerated to LCG. To generate a real value U_i between 0 and 1, one can use either $U_i = X_i/p$ or, as recommended by Deng and Xu [7], $U_i = (X_i + 0.5)/p$.

The characteristic polynomial corresponding to the MRG in (2) has the following form:

$$f(x) = x^k - \alpha_1 x^{k-1} - \cdots - \alpha_k. \tag{3}$$

The maximum-period of such an MRG is $p^k - 1$, which is achieved if and only if its characteristic polynomial $f(x)$ is a primitive polynomial modulo p. A maximum-period MRG of order k enjoys a nice equi-distribution property up to k dimensions. That is, every m-tuple of integers between 0 and $p-1$ appears exactly the same number of times over the entire period $p^k - 1$ with the exception that the all-zero tuple appears one time less. For further details, please see Lidl and Niederreiter [17].

When the order k becomes large, we need to overcome two major barriers when considering an MRG. One is the efficient implementation problem and the other is the efficient searching problem. An MRG of the general form (2) is not as efficient as

an LCG, because it demands many more multiplication operations than an LCG. To improve the efficiency, many researchers considered a special form of MRG that has only two non-zero terms. Please see, for example, Grube [11] and L'Ecuyer and Blouin [13]. Deng and Lin [6] proposed a special form of MRG, called Fast MRG (FMRG), that requires only a single multiplication. Deng and Xu [7] and Deng [3] extended FMRG and introduced a new class of MRGs, called DX-*k*-*s* generators, as follows:

1. DX-*k*-1 ($\alpha_1 = 1, \alpha_k = B$),
$$X_i = X_{i-1} + BX_{i-k} \bmod p, \quad i \geqslant k. \tag{4}$$
2. DX-*k*-2 ($\alpha_1 = \alpha_k = B$),
$$X_i = B(X_{i-1} + X_{i-k}) \bmod p, \quad i \geqslant k. \tag{5}$$
3. DX-*k*-3 ($\alpha_1 = \alpha_{\lceil k/2 \rceil} = \alpha_k = B$),
$$X_i = B(X_{i-1} + X_{i-\lceil k/2 \rceil} + X_{i-k}) \bmod p, \quad i \geqslant k. \tag{6}$$
4. DX-*k*-4 ($\alpha_1 = \alpha_{\lceil k/3 \rceil} = \alpha_{\lceil 2k/3 \rceil} = \alpha_k = B$),
$$X_i = B(X_{i-1} + X_{i-\lceil k/3 \rceil} + X_{i-\lceil 2k/3 \rceil} + X_{i-k}) \bmod p, \quad i \geqslant k. \tag{7}$$

The notation $\lceil x \rceil$, denoting the ceiling function of a real number x, returns the smallest integer $\geqslant x$. For DX-*k*-*s*, k is the order of the generator and s is the number of non-zero terms with the same coefficient B. Because a DX generator requires only one multiplication and few additions to compute the recurrence equation, there are efficient implementations that are almost as fast as the traditional LCGs.

Alanen and Knuth [1] and Knuth [12] described some conditions for a polynomial to be a primitive polynomial. However, it is difficult to check their conditions directly in practice, especially when the values of k and p are large. Alternatively, Deng [2] proposed an efficient algorithm that bypasses the difficulty of factoring a large number and provided an early exit strategy for a failed search to achieve a better efficiency. We remark that the idea of bypassing factoring a large number was first suggested by L'Ecuyer et al. [14]. To bypass the factorization problem, for a given prime k, Deng [2] proposed to find a prime p such that $R(k,p) \equiv (p^k - 1)/(p - 1)$ is a prime number. Such $R(k,p) = (p^k - 1)/(p - 1)$ is termed by Deng [2] as a generalized Mersenne prime (GMP). The idea is based on the fact that the problem of prime number checking is much easier than the problem of factorization.

3. Parallel random number generators based on LCG and MRG

Using LCGs and MRGs as backbone generators to design PRNGs, as mentioned before, the approaches (a) and (b) for designing PRNGs by choosing different moduli and additive constants for different processors, respectively, are not recommended. Thus, it remains to discuss the approaches of choosing (c) starting seeds and (d) multipliers.

3.1. Changing starting seeds for LCGs

Choosing different starting seeds with some efficient skip-ahead schemes for different processors/processes is a popular approach in designing parallel generators. It only uses a single RNG but carefully selects a distinct subsequence for each processor/process. The Lehmer trees method proposed by Frederickson et al. [8] is a typical method of this approach, in which two LCGs were used:

$$L(X) = B_L X \bmod p \quad \text{and} \quad R(X) = B_R X \bmod p, \tag{8}$$

where $L(X)$ is used to generate a new starting seed and $R(X)$ is used when a random variate is needed. If the maximum number of random numbers is known, say n, a simpler method is to use the starting seed $B^{kn} X_0 \bmod p$ for the kth processor. This strategy was used in the random number generator of the NAS parallel benchmarks. Because of the short period of subsequences generated by LCGs, choosing different starting seeds will further shorten the lengths of non-overlapping subsequences. Hence, we do not recommend any method of changing seeds for LCGs.

3.2. Changing different multipliers for LCGs

Choosing different multipliers for different processors or processes can avoid the problems of inter-stream correlation and possible overlapping subsequences. The crucial point of using this approach is: how to quickly and automatically produce different multipliers for which all the corresponding generators are of the maximum-period.

To solve this problem, Deng et al. [5] proposed a systematic leapfrog method to automatically choose different multipliers of LCG generators that each achieves the maximum-period. The idea is that, given a primitive element B modulo p, one can produce a new primitive element by $B_{\text{new}} = B^r \bmod p$ if r is relatively prime to $p - 1$. Its theoretical foundation is based on the following well-known result in number theory. (See, e.g., Niven and Zuckerman [20].)

Theorem 1. *Let p be a prime number and B be a primitive element modulo p. For any positive integer C less than p, there exists an integer r, $1 \leqslant r \leqslant p-1$, such that $C = B^r \bmod p$. Furthermore, C is also a primitive element modulo p if and only if $\gcd(r, p-1) = 1$.*

Deng et al. [5] proposed to use another LCG to generate systematically and randomly a sequence of such r's satisfying the condition that every generated r is relatively prime to $p-1$. More specifically, we first choose an R that is a primitive element modulo $p-1$ and is relatively prime to $p-1$. For simplicity, setting the initial value $r_0 = 1$, we can then generate a sequence of such r's by the following recursive equation:

$$r_i = R\, r_{i-1} \bmod (p-1), \quad i \geqslant 1. \tag{9}$$

It is straightforward to see, by (9), that one can produce different r_i's (within the period of the LCG in (9)) that are relatively prime to $p-1$. Deng et al. [5] further proposed to choose p to maximize the generating period of the LCG in (9).

The systematic leapfrog method proposed by Deng et al. [5] is summarized as follows:

1. Whenever a new processor is initiated, generate a new r by

$$r_{\text{new}} = R\, r_{\text{old}} \bmod (p-1).$$

2. Calculate a new multiplier B for the new processor by

$$B_{\text{new}} = B^{r_{\text{new}}} \bmod p.$$

3. New processor will use a new LCG with coefficient B_{new} as

$$X_i = B_{\text{new}} X_{i-1} \bmod p, \ i \geqslant 1.$$

As an example, choose $p = 2^{31} - 69 = 2{,}147{,}483{,}579$ as the prime modulus, the multiplier $B = 1{,}747{,}834{,}819$ as a primitive element modulo p, and $R = 693{,}352{,}593$ as a primitive element modulo $p-1$. Here the prime p is chosen so that $p - 1 = 2Q$ and Q $(= 1{,}073{,}741{,}789)$ is a prime number. Set the initial value $r_{\text{old}} = r_0 = 1$. With these setups, we can generate a new r by

$$r_{\text{new}} = 693{,}352{,}593 r_{\text{old}} \bmod 214{,}7483{,}578.$$

With $r_{\text{old}} = 1$, we have $r_{\text{new}} = 693{,}352{,}593$. Next, we calculate a new B by

$$B_{\text{new}} = 1{,}747{,}834{,}819^{r_{\text{new}}} \bmod 2{,}147{,}483{,}579 = 315{,}852{,}573.$$

Then, we have a new LCG generator for the new processor as

$$X_i = 315{,}852{,}573 X_{i-1} \bmod 2{,}147{,}483{,}579, \quad i \geqslant 1.$$

Repeating the above procedure, one can quickly and randomly construct $Q - 1 = 1{,}073{,}741{,}788$ different LCGs, each with a maximum-period of 2,147,483,578.

3.3. Changing starting seeds for MRGs

Let $f(x) = x^k - \alpha_1 x^{k-1} - \cdots - \alpha_k$ as in (3) be a primitive polynomial and

$$\mathbf{M}_f = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ & \cdot & \cdot & \cdots & \cdot \\ 0 & 0 & 0 & \cdots & 1 \\ \alpha_k & \alpha_{k-1} & \cdot & \cdot & \alpha_1 \end{pmatrix} \tag{10}$$

be the companion matrix of $f(x)$.

Among several important applications of the companion matrix $\mathbf{M}_f$ to MRG, we show below that, when k is small, $\mathbf{M}_f$ can be used to compute efficiently the seeds that are far apart and to generate other maximum-period MRGs quickly.

To generate the k-dimensional seed vector that is "d-apart" from the initial seed vector $\mathbf{x}_{\text{old}} = (X_0, X_1, \ldots, X_{k-1})'$, we use the following:

$$\mathbf{x}_{\text{new}} = \mathbf{M}_f^d\, \mathbf{x}_{\text{old}} \bmod p.$$

As already mentioned in L'Ecuyer et al. [15], the above jump-ahead matrix $\mathbf{M}_f^d \bmod p$ can be computed efficiently by the divide-and-conquer algorithm using the following recursion:

$$\mathbf{M}_f^d \bmod p = \begin{cases} \mathbf{M}_f, & d = 1, \\ (\mathbf{M}_f^{d/2} \bmod p) \times (\mathbf{M}_f^{d/2} \bmod p) \bmod p, & d > 1,\ d \text{ is even}, \\ \mathbf{M}_f \times (\mathbf{M}_f^{d-1} \bmod p) \bmod p, & d > 1,\ d \text{ is odd}. \end{cases}$$

When k is small, the matrix $\mathbf{M}_f^d \bmod p$ can be pre-computed in time $O(\log(d))$ and then be saved for later use. However, it is clear that the task of pre-computing and saving the jump-ahead matrix becomes inefficient when k is large, say, $k > 100$.

3.4. Changing different multipliers for MRGs

Another application of the companion matrix $\mathbf{M}_f$ is to construct other primitive polynomials quickly by the following theorem stated in Deng [2].

Theorem 2. *Let $f(x)$ in (3) be a primitive polynomial and $\mathbf{M}_f$ in (10) be the companion matrix of $f(x)$. Define*

$$f_r(x) = \det(x\mathbf{I} - \mathbf{M}_f^r) \bmod p,$$

where r is a nonnegative integer. Then

1. *$f_r(x)$ is a primitive polynomial if and only if r is relatively prime to $p^k - 1$.*
2. *If $r,\ s$ are relatively prime to $p^k - 1$, then the following two conditions are equivalent:*
   (a) *$f_r(x) = f_s(x)$.*
   (b) *$r = sp^t \bmod (p^k - 1)$, for some integer $t \geqslant 0$.*

Similar statements to the above theorem can also be found in Golomb [10] and Zierler [22] concerning the equivalence relation among decimation of M-sequences. While the first part of this theorem gives a simple verification condition for a new primitive polynomial, the second part is useful for seeing if two primitive polynomials are identical. This theorem in theory is useful in finding all primitive polynomials, but it has no control over the number of non-zero terms in the primitive polynomials generated, a criterion crucial for computational efficiency. Moreover, as discussed earlier, calculating powers of a matrix and the corresponding determinants can be time-consuming or even infeasible when the order k is large.

For MRG, Deng [2] proposed an automatic generating method to construct many maximum-period MRGs from a single MRG. Based on his method, we propose a more efficient way than the method described above for generating primitive polynomials from an already known primitive polynomial in the next section.

4. Automatic generating method for parallel MRGs

4.1. Constructing MRGs with different multipliers

Following Deng [2], we denote MRG-k-s as a class of maximum-period MRGs with order k and s non-zero terms in their linear congruential equations. If $f(x)$ is an irreducible polynomial, then $c^{-k}f(cx)$ and $x^k f(c/x)$ are also irreducible polynomials for any non-zero constant c. Using this fact, Deng [2] gave the following theorem.

Theorem 3. *Let $R(k,p) = (p^k - 1)/(p - 1)$ be a GMP and c be any non-zero integer. Let $f(x)$ in (3) be a primitive polynomial and define*

$$G(x) = c^{-k}f(cx) = x^k - G_1 x^{k-1} - G_2 x^{k-2} - \cdots - G_k \bmod p, \tag{11}$$

$$H(x) = -\alpha_k^{-1} x^k f(c/x) = x^k - H_1 x^{k-1} - H_2 x^{k-2} - \cdots - H_k \bmod p, \tag{12}$$

where $G_j = c^{-j}\alpha_j \bmod p$ and $H_j = -\alpha_k^{-1}\alpha_{k-j}c^j, \bmod p$, for $j = 1, 2, \ldots, k,\ \alpha_0 = -1$. If the constant term $G_k\ (= c^{-k}\alpha_k)$ is a primitive element modulo p, then both $G(x)$ and $H(x)$ are kth degree primitive polynomials.

As mentioned in Deng [2], there are two advantages in considering $G(x)$ and $H(x)$: (i) they can be calculated very efficiently from the polynomial $f(x)$ and (ii) both of them have exactly the same number of non-zero terms as that in $f(x)$. Therefore, if $f(x)$ is the characteristic polynomial of a DX-k-s generator, then the induced generator with characteristic polynomial $G(x)$ or $H(x)$ has only a small number of non-zero terms, thus can be implemented very efficiently in practice.

Based on Theorem 3, many maximum-period MRGs (corresponding to $G(x)$ or $H(x)$) can be constructed quickly from a single maximum-period MRG (corresponding to $f(x)$). However, there are some drawbacks here: (i) we still need to check the primitive root condition of the constant term (G_k or H_k) in the generated polynomial; (ii) it appears that the sequential selection of c should be randomized; but then (iii) the random selection of c causes no guarantee for the generated MRGs to be distinct.

Next, we propose an automatic generating method to overcome these drawbacks.

4.2. Automatic generating method (AGM) algorithm

For efficiency consideration, we choose a DX-*k*-*s* as the backbone generator for constructing a sequence of maximum-period MRGs. Thus, for the maximum-period, the constant term α_k $(= B)$ in the chosen backbone DX generators must be a primitive element modulo p. Then, to automatically generate a sequence of maximum-period MRGs, it suffices to generate a sequence of constants c's such that $c^{-k}B$ (or equivalently, its inverse c^kB^{-1}) is also a primitive element modulo p by Theorem 3.

Since B is a primitive root modulo p, using Theorem 1, any non-zero c can be written as $c = B^d \bmod p$ for some d. Then, producing a sequence of primitive roots of the form

$$c^kB^{-1} = B^{kd-1} \bmod p$$

means we need to find a sequence of exponents d's such that

$$\gcd(kd - 1, p - 1) = 1.$$

Denote such a sequence by $\{d_n\}$. To generate $\{d_n\}$, we first generate a sequence $\{r_n\}$ in which r_n is relatively prime to $p - 1$ for every n by (9). For each r_n, let d_n satisfy

$$kd_n - 1 = r_n \bmod (p - 1),$$

since $\gcd(kd_n - 1, p - 1) = 1$. Thus, we can obtain d_n by $d_n = k^{-1}(r_n + 1) \bmod (p - 1)$ and then c_n by $c_n = B^{d_n} \bmod p$. Here, the notation k^{-1} stands for the inverse of k modulo $p - 1$. We write $k^{-1} = k^* \bmod (p - 1)$ if $k^*k = 1 \bmod (p - 1)$. When k is a prime number and it is not a factor of $p - 1$, $k^{-1} \bmod (p - 1)$ always exists. For a given k, we can certainly put this restriction on p, but instead, we propose choosing a prime p such that $p = 2Q + 1$ and Q is a prime number. Such Q is called a Sophie–Germain prime number. Since k must be an odd prime number, k and $p - 1$ are relatively prime if $k \neq Q$. In practice, the order k usually is much smaller than Q, hence $k \neq Q$. Consequently, $k^{-1} \bmod (p - 1)$ always exists if we choose a p such that Q $(= (p - 1)/2)$ is a Sophie–Germain prime number. Another benefit of this approach is that the maximum-period, $Q - 1$, of the LCG in (9) can be achieved.

Thus, we will consider only the prime number p for which $Q = (p - 1)/2$ is a Sophie–Germain prime hereafter.

AGM Algorithm. Let $R(k, p) = (p^k - 1)/(p - 1)$ be a GMP and $f(x)$ in (3) be a primitive polynomial corresponding to a DX-*k*-*s* generator in which the non-zero coefficient is B as in Eqs. (4)–(7). Let R be a primitive element modulo $p - 1$ with $\gcd(R, p - 1) = 1$. The following procedure will randomly generate a sequence of maximum-period MRGs:

1. Whenever a new processor is initiated, generate r_n by

$$r_n = Rr_{n-1} \bmod (p - 1), \quad n \geqslant 1 \text{ with } r_0 = 1.$$

2. Calculate d_n and then c_n for the new processor by

$$d_n = k^{-1}(r_n + 1) \bmod (p - 1) \quad \text{and} \quad c_n = B^{d_n} \bmod p.$$

3. With the given c_n, compute the primitive polynomial $G(x)$ or $H(x)$ by

$$G(x) = c_n^{-k} f(c_n x) \bmod p,$$
$$H(x) = -B^{-1} x^k f(c_n/x) \bmod p.$$

4. The new processor can use the newly constructed maximum-period MRG of order k corresponding to the characteristic polynomial $G(x)$ or $H(x)$ as follows:

$$X_i = G_1 X_{i-1} + \cdots + G_k X_{i-k} \bmod p,$$
$$X_i = H_1 X_{i-1} + \cdots + H_k X_{i-k} \bmod p.$$

5. Tables of DX-*k*-*s* and MRG-*k*-*s* generators

To construct a sequence of distinct maximum-period MRGs, we first find some DX-*k*-*s* generators and use them as the backbone generators, where k ranges from 101 to 4001 and $s = 1, 2, 3, 4$.

5.1. List of DX-k-s generators

Finding a DX-*k*-*s* generator can be tedious and time-consuming when the order k is large. For a given prime k, we search for a prime $p = 2^{31} - w$ with restrictions that $Q = (p - 1)/2$ and $(p^k - 1)/(p - 1)$ are also prime numbers. We then search for B in DX-*k*-*s* generators with a certain bound on B to achieve portability and efficiency. Following IEEE double precision standard, one can use the following upper bounds on B for portable DX-*k*-*s* generators:

$$B < 2^d, \quad \text{where } d = 20 \text{ when } s = 1, 2; \text{ and } d = 19 \text{ when } s = 3, 4. \tag{13}$$

Because of the limited time and space, we only consider the smallest prime k in each interval of 100 starting from $k = 101$ to 4001. For each such k, we are able to find the largest prime $p < 2^{31}$ for which both $Q = (p-1)/2$ and $(p^k - 1)/(p-1)$ are also prime numbers. Using the efficient algorithm proposed in Deng [2], we have found some DX-k-s generators and list them in Table 1.

The first column in Table 1 is the order k, the second column is the w for which $p = 2^{31} - w$ given in the third column is a prime, the fourth column is the number of digits for the maximum-period $p^k - 1$, the fifth column is the primitive element R modulo $(p-1)$ for AGM, and the final four columns are the values of B we found for the DX-k-s generators corresponding to $s = 1, 2, 3, 4$, respectively. It is possible to choose the same primitive element R modulo $(p-1)$ for various prime p. For simplicity, we choose the same R for a group of 10 p's listed in Table 1.

Choosing an appropriate size for k from Table 1 depends on the type of applications. A smaller value of k requires a smaller memory space and a quicker initialization. On the other hand, a larger value of k is suitable when the high-dimensional equi-distribution property is a major concern.

The actual searching time needed for a DX-k-s generator is fairly random and the most important factor concerning the searching time is the size of k. For $k = 101$, the searching time can be as short as a few seconds, whereas for $k = 4001$, the required searching time ranges widely from a few hours to a few days on PCs with Pentium 4 at 2.8 GHz.

The implementation of the DX-k-s generators is easy and a C program can be found in the web site http://www.cs.memphis.edu/~dengl/dx-rng/.

Table 1
List of prime modulus p, primitive element R, and $B < 2^d$ for DX-k-s, $s \leqslant 4$.

k	w	$p = 2^{31} - w$	$\log_{10}(p^k - 1)$	R	$B(s=1)$	$B(s=2)$	$B(s=3)$	$B(s=4)$
101	82,845	2,147,400,803	942.5	25,533	1,048,575	1,048,498	524,190	524,288
211	841,329	2,146,642,319	1969.0	25,533	1,048,216	1,047,751	524,256	523,715
307	52,545	2,147,431,103	2864.9	25,533	1,046,286	1,048,079	524,121	524,181
401	57,189	2,147,426,459	3742.1	25,533	1,048,334	1,048,222	523,843	522,593
503	174,489	2,147,309,159	4693.9	25,533	1,048,331	1,047,794	523,798	524,161
601	1,327,485	2,146,156,163	5608.3	25,533	1,043,822	1,047,906	521,759	522,311
701	220,665	2,147,262,983	6541.7	25,533	1,046,874	1,047,056	522,314	522,625
809	2,010,789	2,145,472,859	7549.2	25,533	1,044,987	1,036,488	522,692	522,901
907	4,400,889	2,143,082,759	8463.3	25,533	1,047,699	1,044,229	516,836	523,609
1009	2,368,869	2,145,114,779	9415.4	25,533	1,047,683	1,047,799	522,555	523,048
1103	7,316,361	2,140,167,287	10,291.5	25,239	1,047,649	1,048,009	521,115	519,187
1201	1,113,705	2,146,369,943	11,207.4	25,239	1,044,395	1,048,136	522,631	524,018
1301	1,070,901	2,146,412,747	12,140.6	25,239	1,047,834	1,046,992	524,187	521,141
1409	4,320,189	2,143,163,459	13,147.5	25,239	1,046,153	1,046,464	524,103	523,743
1511	2,771,205	2,144,712,443	14,099.7	25,239	1,048,520	1,039,829	519,262	519,614
1601	368,961	2,147,114,687	14,940.3	25,239	1,048,172	1,047,402	522,467	522,321
1709	1,032,441	2,146,451,207	15,947.9	25,239	1,043,790	1,044,769	518,391	523,880
1801	5,789,241	2,141,694,407	16,804.7	25,239	1,045,648	1,040,074	517,427	518,459
1901	267,321	2,147,216,327	17,739.9	25,239	1,047,198	1,042,940	512,463	520,954
2003	44,961	2,147,438,687	18,691.8	25,239	1,043,074	1,039,648	519,539	523,999
2111	3,536,385	2,143,947,263	19,698.2	32,809	1,048,318	1,045,032	517,247	522,842
2203	6,043,089	2,141,440,559	20,555.5	32,809	1,041,675	1,047,569	523,406	523,680
2309	340,185	2,147,143,463	21,547.3	32,809	1,046,953	1,041,010	524,185	511,205
2411	9,256,449	2,138,227,199	22,494.8	32,809	1,046,643	1,041,950	524,025	524,010
2503	13,539,249	2,133,944,399	23,351.0	32,809	1,048,517	1,046,984	521,989	522,846
2609	8,811,681	2,138,671,967	24,342.4	32,809	1,033,756	1,046,240	517,271	522,508
2707	1,113,585	2,146,370,063	25,260.9	32,809	1,048,221	1,045,429	522,221	519,553
2801	1,095,609	2,146,388,039	26,138.1	32,809	1,047,344	1,044,242	524,187	522,942
2903	14,055,825	2,133,427,823	27,082.3	32,809	1,048,504	1,039,239	523,893	523,072
3001	3,058,401	2,144,425,247	28,003.3	32,809	1,048,008	1,047,926	523,804	523,972
3109	6,741,129	2,140,742,519	29,008.7	33,455	1,045,716	1,045,095	519,235	521,537
3203	4,718,889	2,142,764,759	29,887.1	33,455	1,047,794	1,045,174	522,472	520,906
3301	14,881,185	2,132,602,463	30,794.7	33,455	1,048,195	1,047,412	520,728	524,261
3407	6,243,009	2,141,240,639	31,789.6	33,455	1,040,788	1,036,658	522,501	520,394
3511	1,412,961	2,146,070,687	32,763.4	33,455	1,044,201	1,048,511	516,578	519,482
3607	1,026,585	2,146,457,063	33,659.5	33,455	1,044,732	1,045,641	515,337	520,749
3701	11,576,625	2,135,907,023	34,528.8	33,455	1,045,455	1,034,828	509,071	516,104
3803	32,058,129	2,115,425,519	35,464.5	33,455	1,037,342	1,044,969	517,351	519,156
3907	17,381,649	2,130,101,999	36,446.1	33,455	1,042,792	1,046,828	512,332	518,758
4001	4,412,481	2,143,071,167	37,333.5	33,455	1,044,560	1,031,978	516,937	520,508

5.2. Finding MRG-k-s generators

For each DX-*k*-*s* generator, $s = 1, 2, 3, 4$, we can apply the proposed AGM algorithm to automatically and quickly construct a sequence of $G(x)$'s or $H(x)$'s. The most simple cases are when $s = 1$ and 2, for which both $G(x)$ and $H(x)$ have only two non-zero terms, and thus the corresponding MRG generators are very efficient.

For illustration purpose, let us take $k = 4001$ as a concrete example. For $k = 4001$, we find from the last row of Table 1 that $p = 2^{31} - 4{,}412{,}481 = 2{,}143{,}071{,}167$. One can easily verify that $Q = (p-1)/2 = 1{,}071{,}535{,}583$ is indeed a Sophie–Germain prime. From the column under $s = 2$, we find $B = 1{,}031{,}978$, which implies that the following polynomial of degree 4001

$$f(x) = x^{4001} - 1{,}031{,}978\, x^{4000} - 1{,}031{,}978$$

is a primitive polynomial. Consequently, the corresponding DX-4001-2 generator is

$$X_i = 1{,}031{,}978(X_{i-1} + X_{i-4001}) \bmod 2{,}143{,}071{,}167, \quad i \geqslant 4001,$$

with the period length of $10^{37{,}333.5}$.

To apply AGM algorithm, we find a primitive element $R = 33{,}455$ modulo 2,143,071,166 from Table 1. We then use

$$r_n = 33{,}455\, r_{n-1} \bmod 2{,}143{,}071{,}166, \quad r_0 = 1,$$

to generate a random sequence $\{r_n\}$ that all of the elements are relatively prime to 2,143,071,166. In total, there are $Q - 1 = 1{,}071{,}535{,}582$ different r_n that can be generated using the above LCG. With each r_n, we can calculate d_n and then c_n easily by $d_n = k^{-1}(r_n + 1) \bmod (p-1)$ and $c_n = B^{d_n} \bmod p$. With c_n, we then compute the corresponding $G(x)$ or $H(x)$ to construct a new MRG of the maximum-period. Consequently, we have constructed many MRG-4001-2 generators, each with the maximum-period of $10^{37{,}333.5}$ and enjoying the nice 4001-dimensional equi-distribution property. The generated r_n, c_n, coefficients of $G(x)$ and $H(x)$ of the first 30 iterations of AGM algorithm are listed in Table 2.

While it may take a long time to find a single DX-4001-2 generator, the total time to construct all MRG-4001-2 generators given in Table 2 was less than 1 second.

From the first row of Table 2, we find $G_1 = 538{,}038{,}547$, $G_{4001} = 466{,}567{,}840$ and its corresponding MRG-4001-2 generator,

$$X_i = 538{,}038{,}547X_{i-1} + 466{,}567{,}840X_{i-4001} \bmod 2{,}143{,}071{,}167, \quad i \geqslant 4001,$$

Table 2
List of the first 30 iterations of AGM algorithm for constructing MRG-4001-2.

n	r_n	c_n	G_1	G_{4001}	H_{4000}	H_{4001}
1	33,455	271,596,069	538,038,547	466,567,840	377,755,423	784,137,450
2	1,119,237,025	869,504,607	550,884,537	478,847,729	657,202,932	1,753,090,457
3	335,259,023	442,515,096	1,566,662,175	187,227,285	1,296,770,865	1,857,614,561
4	1,399,202,787	104,753,893	1,315,679,652	1,107,629,070	810,654,320	328,178,428
5	1,368,831,313	986,411,888	1,651,288,829	254,685,660	1,154,006,112	1,220,729,562
6	1,106,901,327	173,373,102	2,075,793,756	1,281,741,128	1,384,090,581	906,013,825
7	1,257,217,471	379,485,739	1,845,836,277	409,133,161	1,335,004,469	1,908,485,007
8	295,788,389	1,096,956,795	1,026,567,566	47,651,946	1,418,866,529	1,884,598,511
9	1,040,980,573	33,287,558	1,992,756,369	815,795,955	1,878,987,105	518,749,096
10	1,098,622,215	948,019,578	1,188,384,449	237,670,234	1,538,951,230	966,296,673
11	735,705,925	1,736,486,493	1,922,179,869	783,700,166	1,125,176,950	651,066,012
12	2,012,450,531	1,743,691,697	2,093,852,176	1,148,153,589	2,087,924,570	1,875,568,645
13	1,951,834,715	1,940,817,358	1,962,052,560	788,843,490	1,120,647,280	496,161,642
14	1,395,033,471	1,164,585,786	100,0823,826	2,090,072,189	1,793,563,909	1,630,879,776
15	1,183,990,323	1,242,931,911	1,134,617,119	1,105,902,684	35,171,343	41,708,201
16	11,894,787	1,788,163,231	143,622,906	356,341,728	1,144,623,554	2,016,376,913
17	1,471,933,375	26,911,583	1,391,019,115	1,651,184,672	1,885,722,755	2,126,035,671
18	41,808,277	1,136,237,452	721,007,986	805,583,938	2,123,397,604	2,075,579,134
19	1,413,506,803	348,179,413	1,501,753,337	1,577,403,115	939,249,849	823,191,446
20	2,004,816,575	1,461,887,585	628,558,650	838,743,597	614,541,364	1,013,965,539
21	1,583,305,489	986,509,896	323,113,640	597,147,228	1,099,408,787	354,393,032
22	1,338,195,639	1,967,919,631	1,796,691,331	756,335,575	755,747,818	1,026,021,504
23	578,445,005	217,682,663	1,397,322,851	2,128,793,813	57,165,708	1,852,038,451
24	2,088,084,461	2,121,691,915	911,105,958	499,380,695	656,877,729	1,378,697,530
25	1,317,915,819	1,993,594,908	1,530,936,371	1,926,134,047	458,919,955	495,348,848
26	1,470,626,527	567,138,616	1,419,063,622	618,427,520	642,345,579	974,042,996
27	1,325,702,923	1,241,853,267	98,160,160	1,287,104,077	1,642,600,570	1,341,631,739
28	533,508,595	1,482,853,899	1,268,809,337	2,040,508,059	675,468,491	1,311,372,269
29	1,033,375,277	872,212,602	1,325,677,274	520,430,780	152,482,690	1,229,037,168
30	1,688,913,289	268,268,315	251,241,551	963,812,485	2,074,449,625	1,765,384,041

Table 3
List of prime p, primitive element R, and $B < 2^d$ for DX-k-s, $s \leqslant 4$, $k \leqslant 10,007$.

k	w	$p = 2^{31} - w$	$\log_{10}(p^k - 1)$	R	$B(s=1)$	$B(s=2)$	$B(s=3)$	$B(s=4)$
5003	1,259,289	2,146,224,359	46,686.4	24,349	1,041,088	1,039,973	506,762	487,092
6007	9,984,705	2,137,498,943	56,044.7	24,349	1,046,897	1,015,366	519,071	519,501
7001	610,089	2,146,873,559	65,332.0	24,349	1,026,965	1,014,115	521,869	506,984
8009	5,156,745	2,142,326,903	74,731.1	24,349	1,041,446	1,046,062	519,082	518,174
9001	7,236,249	2,140,247,399	83,983.5	24,349	1,045,508	1,040,383	515,350	523,991
10,007	431,745	2,147,051,903	93,383.7	24,349	1,042,089	1,042,654	515,671	493,723

has the maximum-period of $10^{37,333.5}$. Similarly, we can construct another maximum-period MRG-4001-2 generator from $H(x)$

$$X_i = 377,755,423X_{i-4000} + 784,137,450X_{i-4001} \bmod 214,3071,167, \quad i \geqslant 4001.$$

In practice, if the number of processors is of moderate size, we can pre-compute the coefficients of $G(x)$ and/or $H(x)$ as given in Table 2. If the number of processors is huge or unknown, then we can use the proposed AGM method to produce practically unlimited number of distinct MRGs each with a huge maximum-period.

6. Tables of DX-*k*-*s* generators of large order *k*

Recently, Deng [4] found some prime modulus p for k from 5003 up to 10007 such that $R(k,p) = (p^k - 1)/(p - 1)$ is prime. Following the same procedure as previously discussed, Deng [4] found a list of DX-k-s generators as given in Table 3.

As an example, with $k = 10,007$, we find several DX-10,007 generators each having the period length of $10^{93,383.7}$. We can then use the automatic generating method to quickly find many MRGs of order 10,007 each with the same period length.

Acknowledgements

The authors are grateful to the Editor and an anonymous referee who made many helpful comments and suggestions. Lih-Yuan Deng was supported partially by NSF grant DMS-0805829. The work of Jyh-Jen Horng Shiau was supported in part by the National Research Council of Taiwan, Grant No. NSC95-2118-M-009-006-MY2 and NSC97-2118-M-009-002-MY2.

References

[1] J.D. Alanen, D.E. Knuth, Tables of finite fields, Sankhyā, Series A 26 (1964) 305–328.
[2] L.Y. Deng, Generalized Mersenne prime number and its application to random number generation, in: H. Niederreiter (Ed.), Monte Carlo and Quasi-Monte Carlo Methods 2002, Springer-Verlag, 2004, pp. 167–180.
[3] L.Y. Deng, Efficient and portable multiple recursive generators of large order, ACM Transactions on Modelling and Computer Simulation 15 (1) (2005) 1–13.
[4] L.Y. Deng, Issues on computer search for large order multiple recursive generators, in: S. Heinrich, A. Keller, H. Niederreiter (Eds.), Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer-Verlag, 2007, pp. 251–261.
[5] L.Y. Deng, K.H. Chan, Y. Yuan, Design and implementation of random number generators for multiprocessor systems, International Journal of Modelling and Simulation 14 (4) (1994) 185–191.
[6] L.Y. Deng, D.K.J. Lin, Random number generation for the new century, American Statistician 54 (2000) 145–150.
[7] L.Y. Deng, H.Q. Xu, A system of high-dimensional, efficient, long-cycle and portable uniform random number generators, ACM Transactions on Modelling and Computer Simulation 13 (4) (2003) 299–309.
[8] P. Frederickson, R. Hiromoto, T. Jordan, B. Smith, T. Warnockr, Pseudo-random trees in Monte Carlo, Parallel Computing 1 (2) (1984) 175–180.
[9] S. Geman, D. Geman, Stochastic relaxation, Gibbs distributions and Bayesian restoration of images, IEEE Transactions on Pattern Analysis and Machine Intelligence 6 (1984) 721–741.
[10] S.W. Golomb, Shift Register Sequence, Holden-Day, San Francisco, CA, 1967.
[11] A. Grube, Mehrfach rekursiv-erzeugte Pseudo-Zufallszahlen, Zeitschrift fur Angewandte Mathematik und Mechanik 53 (1973) 223–225.
[12] D.E. Knuth, The Art of Computer Programming, third ed., vol. 2, Addison-Wesley, Reading, MA, 1998.
[13] P. L'Ecuyer, F. Blouin, Linear congruential generators of order $k > 1$, in: 1988 Winter Simulation Conference Proceedings, 1988, pp. 432–439.
[14] P. L'Ecuyer, F. Blouin, R. Couture, A search for good multiple recursive linear random number generators, ACM Transactions on Modelling and Computer Simulation 3 (1993) 87–98.
[15] P. L'Ecuyer, R. Simard, E.J. Chen, W.D. Kelton, An objected-oriented random-number package with many long streams and substreams, Operations Research 50 (6) (2002) 1073–1075.
[16] D.H. Lehmer, Mathematical methods in large-scale computing units, in: Proceedings of the Second Symposium on Large Scale Digital Computing Machinery, Harvard University Press, Cambridge, MA, 1951, pp. 141–146.
[17] R. Lidl, H. Niederreiter, Introduction to Finite Fields and Their Applications, revised ed., Cambridge University Press, Cambridge, MA, 1994.
[18] M. Mascagni, Parallel linear congruential generators with prime moduli, Parallel Computing 24 (1998) 923–936.
[19] M. Mascagni, A. Srinivasan, Algorithm 806: SPRNG: a scalable library for pseudorandom number generation, ACM Transactions on Mathematical Software 26 (2000) 436–461.
[20] I. Niven, H.S. Zuckerman, An Introduction to the Theory of Numbers, third ed., John Wiley and Sons, New York, 1972.
[21] A. Srinivasan, M. Mascagni, D. Ceperley, Testing parallel random number generators, Parallel Computing 29 (2003) 69–94.
[22] N. Zierler, Linear recurring sequences, Journal of SIAM 7 (1959) 31–48.