Recent Developments in Parallel Pseudorandom Number Generation

Michael Mascagni[*] Steven A. Cuccaro[*] Daniel V. Pryor[*]
M. L. Robinson[*]

Abstract
We summarize some of the recent developments of our research group and of other groups in the design and analysis of pseudorandom number generators for massively parallel computers. The three parallelization techniques we will consider in detail for mapping pseudorandom streams onto distinct parallel processes are:

1. Splitting maximal-period generators' full period into nonoverlapping subsequences, one for each parallel process.

2. Finding parameterized families of generators and distributing a unique parameter value for the generator used on each parallel process.

3. Finding generators with statistically similar full-period equivalence classes and distributing one equivalence class to each parallel process.

1 Introduction

The importance of simulation and the Monte Carlo method in scientific computing is considerable. In fact, some of the first large-scale efforts in high-speed computing were Monte Carlo computations. Moreover, Monte Carlo methods represent a large class of embarassingly parallel algorithms that are thought to be ideally suited to modern parallel architectures. Yet, despite their inherent parallelism, these algorithms cannot perform well without high quality parallel pseudorandom number generators (PRNGs). Below, we summarize recent developments in parallel pseudorandom number generation by considering, in turn, three general approaches to providing parallel PRNGs. The first tact is to take a single serial PRNG and split its full cycle into substrings that are to be used by the different parallel processes. The second approach is to formulate a family of PRNGs which depend on a parameter. Each process is then given the same PRNG but a different set of parameter values. Finally, we consider a single PRNG that has full-period cycles that fall into different equivalence classes depending on the initial seed. This PRNG is then seeded appropriately to ensure that each parallel process uses a different equivalence class.

The major problem in parallel PRNG is ensuring that the work done on individual parallel processes is statistically independent from that done on other processes so their combination further reduces the variance of the full parallel computation. Unfortunately, the notion of statistical independence is very difficult to prove under the simplest conditions, even for numbers generated contiguously from a single PRNG. Thus a universal criterion for the independence of parallel generated pseudorandom numbers does not exist. However, for certain applications, criteria for good quality pseudorandom numbers can be found. Below we will present several criteria that lead to provable results of statistical independence for

[*]Algorithms and Applications Research Group, Supercomputing Research Center, 17100 Science Drive, Bowie, Maryland 20715-4300 **USA**

the various methods of parallel PRNG. We also state those results which we feel are true, but no proof exists at this time.

2 Splitting Schemes

We now discus methods for splitting maximal-period generators into nonoverlapping substreams for use on individual processes. Maximal-period generators have periods that are approximately 2^s, where the generator's state (the information needed to generate the next element in the sequence) consists of s bits. We take the point of view that any sequence we choose to split must have the longest possible period given the amount of computer memory required to generate the next element: thus we restrict ourselves to maximal-period generators. In addition, we know that the simple Monte Carlo application of one-dimensional quadrature is made inefficient if the same pseudorandom numbers are reused in the computation. Thus we make the additional intuitive requirement that the splitting scheme we choose must endeavor to prevent overlap between the split substreams.

Thus, we take a top down approach to choosing both a splitting scheme and a serial PRNG appropriate for splitting. There are five properties that we seek of a serial PRNG that make it suitable for splitting. These are:

(P1) the existence of a "fast leap-ahead" algorithm;
(P2) period long enough to be split;
(P3) serial pseudorandomness;
(P4) substream independence;
(P5) and a fast serial implementation.

(P2) is, of course, met by maximal-period generators, of which there are many [5]. Requirement **(P1)** is much more restrictive, and reduces the possibilities considerably. More precisely, **(P1)** requires the existence of an algorithm that permits "leaping" from element x_n directly to element x_{n+k} of the sequence in no more than $O(\log k)$ generation steps. To our knowledge this leads to the consideration of either linear recursive sequences or to the single nonlinear family of inversive congruential generators (ICGs). Among the linear recursive sequences, requirement **(P5)** restricts attention to either the family of linear congruential generators (LCGs) or to the shift-register generators (SRGs). **(P3)** provides an added restriction, as the full-period pseudorandomness properties of maximal-period LCGs and ICGs are maximized when a *prime* modulus is chosen instead of a power-of-two, modulus [11]. **(P4)** is, by far, the most difficult requirement to fulfill, as the notion of substream independence itself is not well defined. However, if we begin with the assumption that **(P4)** is at least partly satisfied by a splitting scheme that forbids substream overlap, we can study the remaining families of generators for their suitability.

A LCG is defined by the sequence:

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

The constants a and c in (1) are residues modulo m and are called the multiplier and additive constant. For m prime, the maximal period is $m-1$ when a is chosen to be a primitive root modulo m[1] for any choice of c. Since choosing $c = 0$ in this case does not change the period nor the serial pseudorandomness properties, it is prudent only to consider prime modulus LCGs of the form:

$$x_n = ax_{n-1} \pmod{m} \tag{2}$$

[1]We say that a is a primitive root modulo a prime, m, if the least τ such that $a^\tau \equiv 1 \pmod{m}$ is $\tau = m-1$.

with a chosen to a primitive root modulo m.

An ICG is defined by the sequence:

$$x_n = a\bar{x}_{n-1} + c \pmod{m}. \tag{3}$$

Equation (3) is similar to equation (1) except that $\bar{x}_{n-1}$ replaces x_{n-1}. By $\bar{x}$ we mean the modulo m multiplicative inverse, i.e. the residue modulo m such that $\bar{x}x \equiv 1 \pmod{m}$ with $\bar{0} = 0$ by definition. Here, as with LCGs, *prime* modulus generators have superior pseudorandomness properties as compared to power-of-two modulus generators. In contrast to prime modulus LCGs, the maximal period of prime modulus ICGs is m. The conditions for choosing a and c to obtain the maximal period are well known, [11], and in all of these generators $c \neq 0$. These generators are not widely used largely due to the relatively high cost of computing modular inverses. However, recent work has shown that careful implementation of Mersenne prime[2] modular inversion can be done quite effectively on a wide variety of high-performance architectures, [2].

Finally, SRGs are defined bit-by-bit via simple three term recursions:

$$x_n = x_{n-s} + x_{n-r} \pmod{2}. \tag{4}$$

Equation (4) requires that $r > s$ bits be stored as state, and provided the characteristic polynomial defined by (4) is primitive[3], it has a maximal period of $2^r - 1$. Tables of primitive trinomials that lead to maximal period sequences like those in (4) are readily available in tables, [3], [6]. Unlike LCGs and ICGs, SRGs are used to generate pseudorandom uniform variates by concatenating the bits produced in (4) in one of two way. The concatenation of l contiguous and nonoverlapping bits from (4) into an l-bit fraction is called the digital multistep or Tausworthe method. If instead each step of (4) is used to collect l bits from the r-bit state register to form an l-bit fraction, this is called the generalized feedback shift-register (GFSR) method. A fast leap-ahead algorithm and provable serial pseudorandomness properties exist for both of these variations of SRGs, [8], [10], [11].

The splitting scheme we propose is designed to enforce nonoverlap of the subsequences in each process while permitting any process to seed a child process using only local information. The scheme is a generalization of the "Lehmer tree", the original concept put forth for producing pseudorandom numbers for asynchronous MIMD parallel machines, [4]. One view of the Lehmer tree is that it seeds a child process by pseudorandomly leaping ahead the seed of the parent. This procedure *cannot* assure the nonoverlap of the subsequences. Moreover, with an increasing number of processes the probability of overlap exponentially approaches 1. We propose a method called recursive halving leap-ahead, where each new leap is half as long as the previous. This ensures the nonoverlapping of subsequences provided that this procedure has not been applied too often to the same family of subsequences. More on this procedure has appeared elsewhere, [1], [8].

Provable substream independence properties for LCGs exist, [8], [11], but they are rather weak in the sense that trivial bounds exist that are superior for substream lengths less than $O(\sqrt{m})$. Thus for this theory to tell us that good quality exists for split LCGs, each process must consume $O(\sqrt{m})$ PRNs. For ICGs, no such provable results exist, but it is conjectured that the situation is the same as for LCGs, [8], [12]. The one bright spot for splitting is SRGs. It is well known that if x_n comes from a maximal-period SRG given

[2] A Mersenne is a prime of the form $m = 2^q - 1$ where q is also prime.

[3] See [6] for a definition.

by (4), then $y_j = x_{n+j\times 2^l}, j = 0, 1, \ldots$, satisfies (4). Thus if we take a SRG with period $P = 2^q - 1$ and form 2^p parallel streams of length L equally spaced by the amount 2^{q-p}, these $2^{q-p} \times L = \lambda$ numbers can be thought of as coming contiguously from a single SRG. This gives us nontrivial results on quality when not the individual stream lengths, L, but when λ is $O(\sqrt{P})$, [8].

3 Parameterized Generators

We now discuss using parameterized maximal-period generators as a way to furnish each parallel process with its own unique parameterized generator. This approach allows each process to execute the same general algorithm, and hence the same piece of code, with the exception that some parameter passed in initialization is varied from process to process. Theory for this strategy exists for all of the linear recursive sequences considered for splitting. Unfortunately, theory for the analysis of the nonlinear inversive congruential generator does not currently permit this type of analysis except in the recently analyzed case of so-called explicit ICGs. Thus we restrict our discussion to parameterized maximal-period LCGs with prime modulus in order to simplify the presentation.

Since we wish to utilize the same algorithm in each processor, we cannot choose to parameterize the modulus; and since we only consider prime modulus LCGs with zero additive constant, we are left only to consider parameterization via the multiplier. Since maximal-period LCGs must use primitive roots as multipliers, the choice of modulus determines the distribution of available primitive roots. In fact, if a, and α are primitive roots modulo m, then there exists a j with the property $\gcd(j, m-1) = 1$ s.t. $\alpha \equiv a^j \pmod{m}$. A good measure of substream independence in this case is the full-period exponential sum correlation. Given two period $m-1$ streams of modulo m residues, x_n and y_n, the full-period exponential sum is given by:

$$C(j) = \sum_{i=0}^{m-1} e^{\frac{2\pi i}{m}(x_i - y_{i+j})}. \tag{5}$$

The above measures the uniformity in the distribution of differences between all offsets of the sequences. It is a known consequence of the Riemann hypothesis over finite fields that if $x_n = a x_{n-1} \pmod{m}$ and $y_n = a^j y_{n-1} \pmod{m}$ are maximal period LCGs, then, [9], [13]:

$$|C(j)| < (j-1)\sqrt{m}. \tag{6}$$

Thus if we want to have a collection of LCGs that minimizes (6) over all pairs of primitive roots, choosing m to be Fermat, $m = 2^{2^n} + 1$, or Sophie Germain, $m = 2q + 1$ with q also prime, we obtain optimal primitive root distributions due to the fact that these primes have the largest fraction of primitive roots among their modular residues.

4 Equivalence Classes

Finally we discuss the use of submaximal-period generators that have their state space divided into equal length equivalence classes. An example of such a generator is the following linear recursive sequence:

$$x_n = x_{n-5} + x_{n-17} \pmod{2^{32}}.^4 \tag{7}$$

[4]This is the generator currently used in the Thinking Machines Corporation's Connection Machine Scientific Subroutine Library for the CM-2, CM-200, and CM-5 machines.

This generator has a period of $P = (2^{17}-1)\times 2^{31}$ when at least one of the 17 32-bit numbers in its state is odd (i.e. the least significant bit (LSB) of the 17 32-bit numbers cannot be all zeroes), [5]. One can count to find that the number of such full-period state elements is $(2^{17}-1)\times 2^{31\times 17}$. Since each of these full-period state elements must lie in one period P cycle, there are $\frac{(2^{17}-1)\times 2^{31\times 17}}{P} = 2^{31\times 16} = 2^{496}$ different equivalence classes of full-period state elements. A simple but elegant use of these equivalence classes is to distribute a seed from a unique equivalence class to each parallel process. This procedure is similar to the previous parameterization technique in that each process uses the same algorithm; however, instead of passing a different parameter to each process, here we pass a different seed that places the generator in a unique equivalence class. To implement this approach a canonical enumeration of the equivalence classes is required along with an algorithm for producing a seed in a given equivalence class.

An important property of the generator in (7) is that modulo 2 the sequence is a maximal-period shift-register sequence with period $2^{17}-1$. Taken modulo 4 (7) has period $2\times(2^{17}-1)$. In general, modulo 2^j (7) has period $2^{j-1}\times(2^{17}-1)$ for all $j>0$. Since the least significant bit of (7) is a maximal period SRG it is known that all nonzero 17-bit seeds occur in the full-period cycle. Thus we can choose any nonzero seed in a canonical form, since they all occur. The generator $x_n = x_{n-5} + x_{n-17} \pmod{2^{32}}$ can be represented via the 17×17 matrix, $\mathbf{A}$:

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 & 1 \\ 1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & 1 & 0 \end{pmatrix}. \tag{8}$$

The first row of $\mathbf{A}$ has 1's in the 5th and 17th column. Looking at (7) modulo 4, we see that $\mathbf{A}^{2^{17}-1}$ fixes the LSB while sending the most significant bit (MSB) half way around its full period. Thus $\mathbf{A}^{2^{17}-1} \pmod 4$ toggles the MSB into one of two choices. We choose the numerically smaller choice as part of our canonical form. Now $\mathbf{A}^{2\times(2^{17}-1)} \pmod 8$ fixes the two LSBs and toggles the MSB, allowing us to choose the numerically smaller as part of the canonical form. This procedure continues by progressively squaring the power of $\mathbf{A}$ used to select the last bit until a unique equivalence class descriptor is found. Some further analysis of this technique rather unexpectedly produces a fast algorithm for producing a seed in a given equivalence class using this enumeration technique. Space prevents further elaboration, [7].

With this enumeration and the fast algorithm for producing seeds in a given equivalence class of this enumeration we have a parallel PRNG for up to 2^{496} parallel processes. In addition, by exploiting the mapping of equivalence class numbers onto a binary tree, a local computation suffices to seed a child process from a parent. Thus this generator provides the basis for a totally reproducible asynchronous MIMD PRNG suitable for demanding application such as neutron transport Monte Carlo. In addition, since the period of each equivalence class and the number of distinct equivalence classes can be modified by choosing a different additive lagged-Fibonacci generator, we believe that this class of generators may prove most versatile for general massively parallel supercomputer applications, [7].

5 Conclusions

We have briefly discussed three general techniques for placing PRNGs on parallel machines. We presented the recursive halving leap-ahead algorithm for splitting and have shown that in the case of SRGs we can analyze certain simple cases with a provable form of substream independence. In the case of parameterized generators, prime modulus LCGs have shown to give optimal substream independence in another sense when certain prime moduli are chosen. Finally, we have shown how to exploit the equivalence class structure of certain easy to implement lagged-Fibonacci generators to provide a totally reproducible asynchronous MIMD PRNG.

References

[1] F. W. Burton and R. L. Page, *Distributed random number generation*, J. Functional Programming, 2 (1992), pp. 203–212.

[2] S. A. Cuccaro and M. Mascagni, *A comparison of modular inversion implementations across parallel supercomputer architectures*, Supercomputing Research Center Technical Report #92-116, 1992.

[3] S. W. Golomb, *Shift register sequences, Revised Edition*, Aegean Park Press, Laguna Hills, CA, 1982.

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

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

[6] R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*, Cambridge University Press, Cambridge, London, New York, 1986.

[7] M. Mascagni, S. A. Cuccaro, D. V. Pryor, and M. L. Robinson, *Analysis of a lagged-Fibonacci pseudorandom number generator*, (in preparation).

[8] M. Mascagni and M. L. Robinson, *Deterministic splitting of a single pseudorandom sequence into parallel subsequences*, (in progress).

[9] ———, *Parameterized pseudorandom number generation for parallel computers*, (in progress).

[10] H. Niederreiter, *Recent trends in random number and random vector generation*, Annals of Operations Research, 31 (1991), pp. 323–346.

[11] ———, *Random number generation and quasi-Monte Carlo Methods*, CBMS-NSF Regional Conference Series in Applied Math, SIAM, Philadelphia, PA, 1992.

[12] ———, *New methods for pseudorandom number and pseudorandom vector generation*, Proceeding of Winter Simulation Conference, 1992.

[13] W. Schmidt, *Equations over finite fields: An elementary approach*, Lecture Notes in Mathematics #536, Springer-Verlag, Berlin, Heidelberg, New York, 1976.