IMPLEMENTATION AND USAGE OF A PORTABLE AND REPRODUCIBLE PARALLEL PSEUDORANDOM NUMBER GENERATOR

DANIEL V. PRYOR
STEVEN A. CUCCARO
MICHAEL MASCAGNI
M. L. ROBINSON

Supercomputing Research Center, I.D.A. Supercomputing Research Center, I.D.A.
17100 Science Drive
Bowie, Maryland 20715-4300 USA

SRC-TR-94-116

March 15, 1994

ABSTRACT. We describe in detail the parallel implementation of an additive lagged-Fibonacci pseudorandom number generator. Elsewhere, [8], it was shown how important properties of these generators lead to an obvious parallel implementation based on deterministic seeding. Seeds are calculated to select one of a large number of full-period cycles, or equivalence classes. This "naïve" implementation is computationally efficient and produces a high-quality, portable, and totally reproducible parallel generator. A drawback of the "naïve" implementation is that it produces sequences that have initial segments of poor quality. In addition, the initial segments from the equivalence classes of different parallel processes are similar. The simplest solution to this problem is rejected due to its computational cost. Instead, a different seeding algorithm is described. The new algorithm can be viewed as an alternative enumeration of the equivalence classes. The new algorithm is computationally more efficient and avoids the quality problems of the "naïve" implementation. We believe that this algorithm solves the reproducibility problem for a far larger class of parallel Monte Carlo applications than had been previously possible. In fact, this generator allows one to write code that is reproducible independent both of the number of processors and the execution order of the parallel processes. Finally, a library of portable C routines is described that implements these ideas. These routines are available from the authors.

1991 *Mathematics Subject Classification.* 65C10, 65Y05.
Key words and phrases. random number generation, parallel computation, Fibonacci generator, MIMD, SIMD, reproducible, portable, implementation.

1. Introduction.

In Knuth's well known exposition on pseudorandom number generation [5], several methods of generation are considered. Among these is the additive lagged-Fibonacci pseudorandom number generator:

$$x_t = x_{t-k} + x_{t-\ell} \pmod{2^m}, \quad \ell > k, \tag{1}$$

which we shall denote as $F(\ell, k, m)$. In Marsaglia's empirical study of pseudorandom number generators [6], this generator was among many considered. Overall, it did well on all of Marsaglia's "stringent" tests, save for the "non-overlapping birthday spacing test." However, Marsaglia noted that by choosing a generator with a large length, ℓ, improvements are seen in this test. It is important to note that while the lagged-Fibonacci generators did have one shortcoming, in general they performed better than linear congruential generators on the "stringent" tests.

This generator has a maximum possible period of $(2^\ell - 1)2^{m-1}$ under certain easy to check conditions, [7, 3]. Given these simple conditions, the maximum possible period is obtained if and only if at least one of the initial ℓ residues modulo 2^m is odd. In other words, if the least significant bits of the seed are all zero, the maximum possible period is not obtained. For this generator there are $(2^\ell - 1)2^{\ell(m-1)}$ seeds that satisfy the condition for giving the maximum possible period. Since each of these seeds is in a cycle of maximum possible period, there must be

$$E = \frac{(2^\ell - 1)2^{\ell(m-1)}}{(2^\ell - 1)2^{m-1}} = 2^{(\ell-1)(m-1)} \tag{2}$$

different maximal or full-period cycles. Each of these full-period cycles will be called an equivalence class (EC).

In a previous paper, [8], the authors described a way to enumerate all E ECs based on the reduction of a seed from a given EC into a single unique seed representative of the entire EC. In addition, an explicit construction of all of the EC representatives was shown to have the form:

$$\begin{array}{cccc|c|c}
\text{m.s.b} & & & & \text{l.s.b.} & \\
b_{m-1} & b_{m-2} & \cdots & b_1 & b_0 & \\
\hline
\square & \square & \cdots & \square & b_{0\ell-1} & x_{\ell-1} \\
\square & \square & \cdots & \square & b_{0\ell-2} & x_{\ell-2} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\square & \square & \cdots & \square & b_{01} & x_1 \\
\hline
0 & 0 & \cdots & 0 & b_{00} & x_0
\end{array} \tag{3}$$

Here the ℓ-bit long vector of least significant bits, b_0, can be computed in advance for $F(\ell, k, m)$. We call this representation of $F(\ell, k, m)$, i.e, the bottom row of zeros and the precomputed column of least significant bits, its "canonical form." This canonical form tableau leaves exactly $(\ell - 1)(m - 1)$ bits left to be specified to select one of the $2^{(\ell-1)(m-1)}$ ECs.

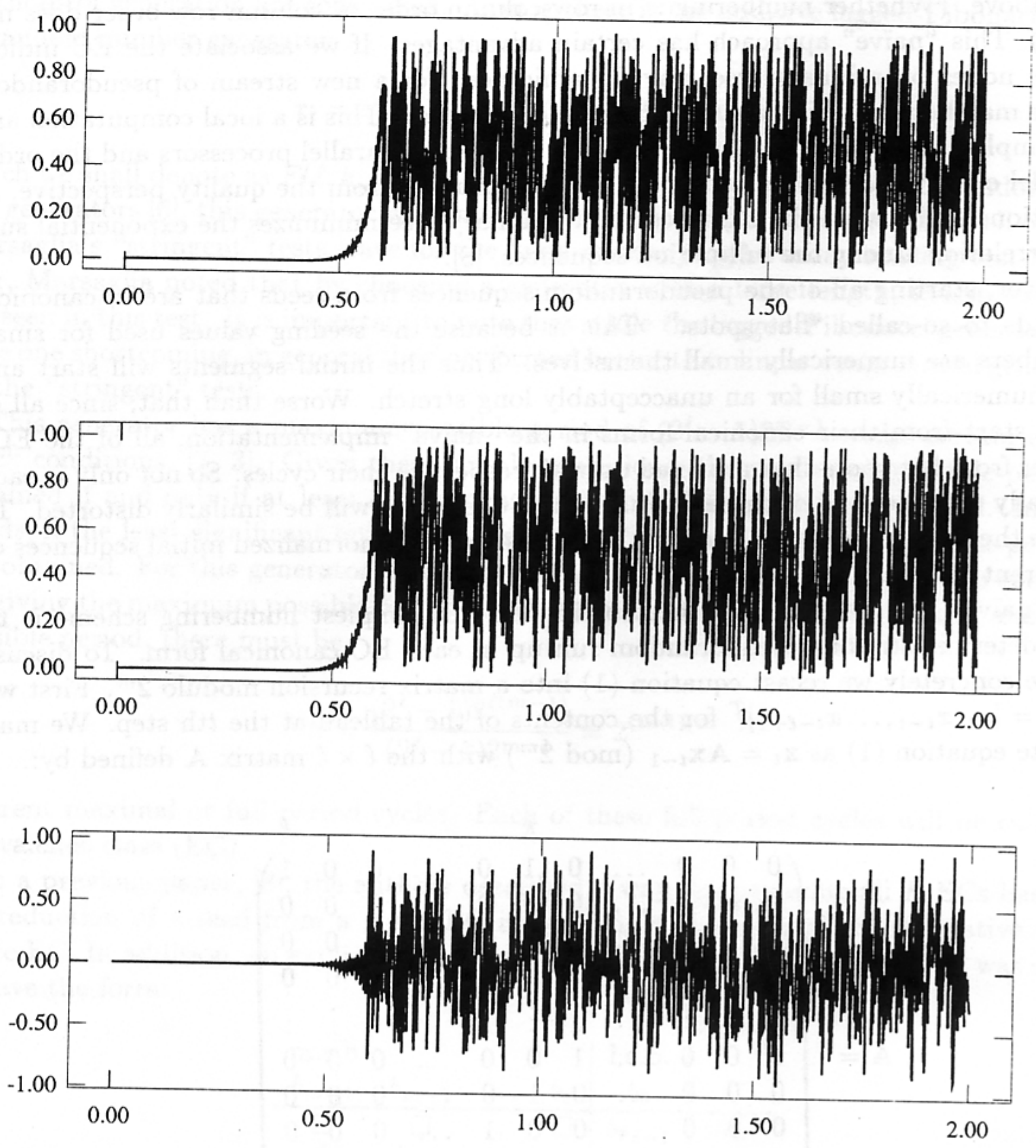
The simplest way to parallelize this generator is to associate the Kth parallel process with EC number K. The numbering is done lexicographically in the rectangle of boxes shown above. (Whether numbering is in row-column order or column-row order does not matter.) This "naïve" approach has certain advantages. If we associate the EC indices with the nodes on a binary tree, any dynamic need for a new stream of pseudorandom numbers may be filled using a successor node on the tree. This is a local computation and can be implemented to be independent of the number of parallel processors and the order of parallel execution, [8]. This approach is quite prudent from the quality perspective, as the relationship between the ECs close on the binary tree minimizes the exponential sum cross-correlation among the full-period sequences, [8].

However, starting all of the pseudorandom sequences from seeds that are in canonical form leads to so-called "flat spots." This is because the seeding values used for small EC numbers are numerically small themselves. Thus the initial segments will start and remain numerically small for an unacceptably long stretch. Worse than that, since all of the ECs start from their canonical forms in the "naïve" implementation, all of the ECs will suffer from flat spots that are lined up with respect to their cycles. So not only is each EC initially distorted, all of those with similar EC number will be similarly distorted. To illustrate the phenomenon of flat spots, Figure 1 shows the normalized initial sequences of two different ECs started from the EC canonical form.

The "naïve" solution to these shortcomings in this simplest numbering scheme is to apply a deterministic but pseudorandom run up in each EC canonical form. To discuss this more concretely we recast equation (1) into a matrix recursion modulo 2^m. First we write $\mathbf{x}_t = [x_t, x_{t-1}, \dots x_{t-\ell+1}]^T$ for the contents of the tableau at the tth step. We may then write equation (1) as $\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1} \pmod{2^m}$ with the $\ell \times \ell$ matrix $\mathbf{A}$ defined by:

$$(4) \qquad \mathbf{A} = \begin{array}{c} \begin{array}{cccccccccccc} & & & & & k & & & & & \ell \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 & 0 & 1 \\ 1 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Thus by run up we mean that if we are in EC number K, we seek to apply some pseudorandom power of $\mathbf{A}$ to the EC canonical form to step it away from the flat spot. We seek a pseudorandom function, $f(\cdot)$, so that we can apply $\mathbf{A}^{f(K)}$ to this EC representative before using it in a computation. Even with efficient powering algorithms and the tabulation of certain special powers of $\mathbf{A}$, this procedure is woefully inefficient. An alternative approach is to renumber the ECs so that the first ECs chosen will not have flat spots and



FIGURES 1A, 1B, AND 1C. The initial sequence from two different ECs started at their EC canonical form. Upper plot (1A) is with EC number $K = 0$. Middle plot (1B) is with $K = 1$. Lower plot (1C) is the difference between 1A and 1B. The units for the abscissa is $\ell \times m$ and the random numbers are scaled to be between 0 and 1.

that neighboring ECs will have very different representatives under the reordering. The description and justification of an efficient algorithm for such a reordering of the ECs is the main purpose of this paper.

The plan of the paper is as follows. In §2 we will describe an alternative seeding algorithm for these generators. This new algorithm will be illustrated for the lagged-Fibonacci generator with parameters $\ell = 17$, $k = 5$, and $m = 32$, that is, $F(17,5,32)$. This generator has $E = 2^{496}$ different ECs. In the alternative seeding algorithm a modulus $2^{31} - 1$ linear congruential generator (LCG) is used to renumber the ECs, eliminating the problem with flat spots. The only property used of the LCG is that it produces all $2^{31} - 2$ non-zero residues modulo $2^{31} - 1$ exactly once in its $2^{31} - 2$ long period. In §3 we describe in detail the incorporation of the algorithm from §2 into a portable library for parallel pseudorandom number generation. We include a discussion on the implementation's use to obtain Monte Carlo results that are independent of the number of parallel processors and on the order of the execution of the independent parallel processes. This is especially useful in applications such as neutronics, where certain types of "splitting" occur. In §4 we summarize the results and propose directions for further work. Finally, in an appendix we include a list of subroutines and their calling sequences for the parallel library available from the authors.

2. The Alternative Seeding Algorithm.

As mentioned above, the generator $F(\ell, k, m)$ has $2^{(\ell-1)(m-1)}$ distinct full-period cycles, or ECs. These can be numbered naturally in an ℓ-digit word, where each "digit" is radix 2^{m-1}. Such a number can be written as $(e_{\ell-2}, e_{\ell-3}, \ldots, e_1, e_0)$, where e_j is the coefficient of $2^{j(m-1)}$. If the generator tableau is in canonical form, then this "word" can be thought of as the rectangle of bits in the upper left hand corner. Each of the $2^{(\ell-1)(m-1)}$ distinct patterns of ones and zeros in that rectangle denotes a distinct EC.

There are two problems to consider in implementing a robust parallel generator which makes use of this cycle structure. (1) How do we avoid the problem of correlation between "neighboring" cycles, at least in the early part of their cycles? and (2) How do we accomplish the spawning of new ECs in such a (reproducible) way as to guarantee that no EC will be spawned more than once? Our approach to these questions resulted in a library of C routines that, for the sake of portability, fixes m to be 32 and gives the user several valid choices for ℓ and k. We will describe the mechanics of the library with $\ell = 17$ and $k = 5$, i.e., $F(17,5,32)$.

Problem (1) is really an acknowledgment that when the 496-bit EC numbers of two cycles differ by only a few bits, their outputs initially will be similar until the effects of the matching bits propagate out of the most significant bit of the tableau. This does not lessen the randomness properties of the individual ECs or their favorable full-period cross-correlation, [8]. Each EC has good full period properties, and flat spots are expected in any sufficiently long random sequence. The problem pointed to here is that with the "natural" numbering of the ECs, we cause the flat spots of the lowest numbered ECs to appear at the very beginning of their cycles, giving the initial appearance of both nonrandomness and high cross-correlation.

For the particular lagged-Fibonacci generator, $F(17,5,32)$, the canonical tableau has the form:

$$\begin{array}{cccc|c|c}
\text{m.s.b} & & & & \text{l.s.b.} & \\
b_{31} & b_{30} & \cdots & b_1 & 0 & \\
\hline
\square & \square & \cdots & \square & 0 & x_{16} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\square & \square & \cdots & \square & 0 & x_{11} \\
\square & \square & \cdots & \square & 1 & x_{10} \\
\square & \square & \cdots & \square & 0 & x_{9} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\square & \square & \cdots & \square & 0 & x_{1} \\
\hline
0 & 0 & \cdots & 0 & 0 & x_0
\end{array} \tag{5}$$

In fact, the least significant bits of this tableau are the same as for the EC tableau for the entire $F(17, 5, \cdot)$ family. Several canonical least significant bit configurations for other families of lagged-Fibonacci generators are listed in Table 1 at the end of this section. The 16-by-31 upper left-hand rectangle of 496 bits is what we can arbitrarily fill to select one of 2^{496} ECs. The bits to be specified are the 31 most significant bits of x_1 through x_{16}. For convenience, let us refer to these bits as the 16-long vector of 31-bit integers $\mathbf{s} = (s_{16}, s_{15}, \ldots, s_2, s_1)$. In the "naïve" implementation, the first EC was filled with $\mathbf{s} = (0, 0, \ldots, 0, 1)$, and so on.

We wish to reorder the ECs. To help us we will employ the LCG:

$$z_t = \Gamma(z_{t-1}) = 7^5 z_{t-1} \pmod{2^{31} - 1} \tag{6}$$

found in [9], to seed the 496 bits. So instead of filling the "first" rectangle as above, we can fill it with $\mathbf{s}_1 = (\Gamma^{15}(1), \Gamma^{14}(1), \ldots \Gamma^2(1), \Gamma(1), 1)$.

Similarly, for EC number 2, we have $\mathbf{s}_2 = (\Gamma^{15}(2), \Gamma^{14}(2), \ldots \Gamma^2(2), \Gamma(2), 2)$. Now the contents of $\mathbf{s}_1$ and $\mathbf{s}_2$ are obviously very different from each other, as they arise from different subsequences taken from the cycle of a full-period LCG. Recall that a full-period LCG with prime modulus, p, has period $p - 1$ and omits only one number. In this case $p = 2^{31} - 1$ is prime, so (6) has period $2^{31} - 2$ and omits 0. Therefore, if we need to seed random number cycles (generators) for each of $N < p$ processes, an initial seed of $\mathbf{s}_n = (\Gamma^{15}(n), \Gamma^{14}(n), \ldots \Gamma^2(n), \Gamma(n), n)$ eliminates the "flat spot" problem and the initial cross correlation of neighboring sequences. Furthermore, this form of initialization can be individualized on a per-run basis by selecting a "global seed," g, that maps each n to a different unique value $\hat{n}$, in the above expression. Thus for any particular computation, g is set at the beginning and $\mathbf{x}_0$ is initialized by first setting $\hat{n} = (n \oplus g) + 1$ (where "$\oplus$" denotes the bitwise exclusive OR operator) and then setting $\mathbf{s}_n = (\Gamma^{15}(\hat{n}), \Gamma^{14}(\hat{n}), \ldots \Gamma^2(\hat{n}), \Gamma(\hat{n}), n)$.[1] Note that the rightmost entry of $\mathbf{s}$ is n rather than $\hat{n}$. There is no reason to prefer one or the other in the present context, but we use n to conform to our usage in subsequent discussions, where it does make a difference. This definition of $\hat{n}$ requires that

[1]Other mappings of n to $\hat{n}$ are certainly possible.