Uniform Random Number Generators for Supercomputers*

Richard P. Brent[1]
Computer Sciences Laboratory
Australian National University

Abstract

We consider the requirements for uniform pseudo-random number generators on modern vector and parallel supercomputers, consider the pros and cons of various classes of methods, and outline what is currently available. We propose a class of random number generators which have good statistical properties and can be implemented efficiently on vector processors and parallel machines. A good method for initialization of these generators is described, and an implementation on a Fujitsu VP 2200/10 vector processor is discussed.

1 Introduction – Requirements

Pseudo-random numbers have been used in Monte Carlo calculations [1, 3, 15] since the pioneering days of Von Neumann [26]. With the increasing speed of vector processors and parallel computers, considerable attention must be paid to the quality of random number generators available in subroutine libraries. A program running on a supercomputer might use 10^8 random numbers per second over a period of many hours (or months in the case of QCD calculations), so 10^{12} or more random numbers might contribute to the result. Small correlations or other deficiencies in the random number generator could easily lead to spurious effects and invalidate the results of the computation.

Applications require random numbers with various distributions (e.g. normal, exponential, Poisson, ...) but the algorithms used to generate these random numbers almost invariably require a good uniform random number generator – see for example [2, 16, 29]. In this paper we consider only the generation of uniformly distributed numbers. Usually we are concerned with *real* numbers u_n which are intended to be uniformly distributed on the interval $[0, 1]$. Sometimes it is convenient to consider *integers* U_n in some range $0 \le U_n < m$. In this case we require $u_n = U_n/m$ to be (approximately) uniformly distributed.

Pseudo-random numbers generated in a deterministic fashion on a digital computer can not be truly random. What is required is that finite segments of the sequence $u_0, u_1, \cdots$ behave in a manner indistinguishable from a truly random sequence. In practice, this means that they pass all statistical tests which are relevant to the problem at hand. Since the problems to which a library routine will be applied are not known in advance, random number generators in subroutine libraries should pass a number of stringent statistical tests (and not fail any) before being released for general use.

A sequence $u_0, u_1, \cdots$ depending on a finite state must eventually be periodic, i.e. there is a positive integer p such that $u_{n+p} = u_n$ for all sufficiently large n. The minimal such p is called the *period.*

Following are some of the more important requirements for a good pseudo-random number generator and its implementation in a subroutine library –

- *Uniformity.* The sequence of random numbers should pass statistical tests for uniformity of distribution. In one dimension this is easy to achieve. Most generators in common use are provably uniform (apart from discretization due to the finite wordlength) when considered over their full period.

- *Independence.* Subsequences of the full sequence $u_0, u_1, \cdots$ should be independent. For example, members of the even subsequence $u_0, u_2, u_4, \cdots$ should be independent of their odd neighbours $u_1, u_3, \cdots$. Thus, the sequence of pairs (u_{2n}, u_{2n+1}) should be uniformly distributed in the unit square. More generally, random numbers are often used to sample a d-dimensional space, so the sequence of d-tuples $(u_{dn}, u_{dn+1}, \ldots, u_{dn+d-1})$ should be uniformly distributed in the d-dimensional cube $[0, 1]^d$ for all "reasonable" values of d (certainly for all $d \le 6$).

- *Long Period.* As mentioned above, a simulation might use 10^{12} random numbers. In such a case the period p must exceed 10^{12}. For many generators there are strong correlations between $u_0, u_1, \cdots$ and $u_m, u_{m+1}, \cdots$,

*Appeared in *Proceedings Fifth Australian Supercomputer Conference* (Melbourne, December 1992), 95–104.

[1]E-mail address: rpb@cslab.anu.edu.au
rpb132 typeset using LaTeX

where $m = p/2$ (and similarly for other simple fractions of the period). Thus, in practice the period should be *much* larger than the number of random numbers which will ever be used.

- *Repeatability.* For testing and development it is useful to be able to repeat a run with *exactly* the same sequence of random numbers as was used in an earlier run [15]. This is usually easy if the sequence is restarted from the beginning (u_0). It may not be so easy if the sequence is to be restarted from some other value, say u_m for a large integer m, because this requires saving the state information associated with the random number generator.

- *Portability.* Again, for testing and development purposes, it is useful to be able to generate *exactly* the same sequence of random numbers on two different machines, possibly with different wordlengths. In practice it will be expensive to simulate a long wordlength on a machine with a short wordlength, but the converse should be easy – a machine with a long wordlength (say $w = 64$) should be able to simulate a machine with a smaller wordlength (say $w = 32$) without loss of efficiency.

- *Disjoint Subsequences.* If a simulation is to be run on a machine with several processors, or if a large simulation is to be performed on several independent machines, it is essential to ensure that the sequences of random numbers used by each processor are disjoint. Two methods of subdivision are commonly used. Suppose, for example, that we require 4 disjoint subsequences for a machine with 4 processors. One processor could use the subsequence $(u_0, u_4, u_8, \cdots)$, another the subsequence $(u_1, u_5, u_9, \cdots)$, etc. For efficiency each processor should be able to "skip over" the terms which it does not require. Alternatively, processor j could use the subsequence $(u_{m_j}, u_{m_j+1}, \cdots)$, where the indices m_0, m_1, m_2, m_3 are sufficiently widely separated that the (finite) subsequences do not overlap. This requires some efficient method of generating u_m for large m without generating all the intermediate values $u_1, \ldots, u_{m-1}$.

- *Efficiency.* It should be possible to implement the method efficiently so that only a few arithmetic operations are required to generate each random number, all vector/parallel capabilities of the machine are used, and overheads such as those for subroutine calls are minimal. This implies that the random number routine should return an array of (optionally) several numbers at a time, not just one.

In Sections 2 and 3 we outline some popular classes of random number generators, and consider to what extent they pass or fail our requirements. Then, in Section 4 we comment on various implementations. In Section 5 we consider which algorithms are suitable for vector and/or parallel machines, and in Section 6 we discuss the specific requirements for several classes of machines (vector, MIMD, SIMD, ...). More detailed comments on a vectorized implementation are given in Section 7.

The important (but often neglected) topic of how to initialize generators is discussed in Section 8.

2 Linear congruential generators

Linear congruential generators were introduced by D. H. Lehmer in 1948 and are still probably the most popular class of generators. An integer sequence (U_n) is defined by an initial value U_0 (the "seed") and the recurrence

$$U_{n+1} = (aU_n + c) \bmod m$$

where $m > 0$ is the "modulus", a is the "multiplier" $(0 < a < m)$, and c is an additive constant.

Often the modulus m is chosen to be a power of 2, say $m = 2^w$ where w is close to the integer wordlength. In this case it is possible to achieve a full period m (provided c is odd and $4|(a-1)$). This does not guarantee a good generator – consider the trivial case $a = c = 1$.

When m is a power of two the low-order bits of U_n do not behave randomly – in fact the low order k bits cycle with a period at most 2^k. This should be enough to make us suspicious of such generators! However, there is hope that the high-order bits behave randomly, so the normalized sequence

$$u_n = U_n/m$$

may be usable as a source of pseudo-random numbers in $[0, 1]$.

To avoid the problem of nonrandomness of low-order bits, the modulus m is sometimes chosen to be a prime number. For example, on a 32-bit machine we could take $m = 2^{31}-1$ or $m = 2^{32}-5$. Provided $U_0 \neq 0$ and a is a primitive root mod m, it is possible to obtain period $p = m-1$, even if $c = 0$. It is sometimes convenient that an exact zero does not occur in the sequence.

There is much theory regarding the best choice of multiplier a for linear congruential generators [16, 27], and an exhaustive search has been performed for certain moduli [8]. Marsaglia [19] pointed out the fundamental weakness of the class of linear congruential generators. If d-tuples

$(u_{dn}, u_{dn+1}, \ldots, u_{dn+d-1})$ of normalized numbers are considered as points in the d-dimensional unit cube, then these points lie on a small number N_d of hyperplanes, far less than would be expected of a truly random sequence with discretization error $O(2^{-w})$. In fact

$$N_d \le (d!m)^{1/d} = O(dm^{1/d}).$$

For example, with $m \le 2^{31}$ and $d = 6$, Marsaglia's bound gives $N_d \le 107$.

The reason for such behaviour is intuitively clear. There are 2^{wd} points in the unit d-cube with coordinates exactly representable as w-bit binary fractions. These points lie on 2^w hyperplanes with separation 2^{-w}. However, a linear congruential generator with period $p \le 2^w$ can give at most 2^w of these points, and it is easy to see that some of these points have a large separation $\Omega(2^{-w/d})$ from their nearest neighbours.

Although Marsaglia's result shows that all linear congruential generators perform poorly in high dimensions, there is still a great difference between the best such generators and generators with poorly chosen multipliers. For example [8], a generator RANDU with $m = 2^{31}$, $a = 65539$, $c = 0$ was used in the IBM Scientific Subroutine Library on System 360/370 computers for many years. The multiplier $a = 2^{16} + 3$ may have been chosen so that multiplication could be performed by a small number of shifts and adds. However, using the relation

$$(a-3)^2 = 0 \bmod m$$

it is easy to see that

$$U_{n+2} - 6U_{n+1} + 9U_n = c' \bmod m$$

and

$$u_{n+2} - 6u_{n+1} + 9u_n = c'' \bmod 1,$$

where c' and c'' are constants. This means that 3-tuples generated by RANDU lie on only 16 planes (separated by distance $118^{-1/2} \simeq 0.092$) in the unit cube.

Surprisingly, generators almost as bad as RANDU are still in use. The problems apparent in the choice of multiplier for these generators can be detected using the "Spectral Test" [16] or by a variety of statistical tests on the distribution of d-tuples, with $d \ge 3$ and a sufficiently fine grid.

3 Generalized Fibonacci generators

The Fibonacci numbers satisfy the recurrence

$$F_n = F_{n-1} + F_{n-2}.$$

However, it is easy to see that the corresponding recurrence

$$u_n = u_{n-1} + u_{n-2} \bmod 1$$

does not give a satisfactory sequence of pseudo-random numbers because the inequality

$$u_{n-2} < u_n < u_{n-1}$$

never holds, even though it would hold with probability 1/6 for a random sequence ([16], ex. 3.2.2.2).

Attempts have been made to generalize the Fibonacci recurrence to obtain "generalized Fibonacci" or "lagged Fibonacci" random number generators [12, 16, 30]. Marsaglia [20] considers generators $F(r, s, \theta)$ which satisfy

$$U_n = U_{n-r} \theta U_{n-s}$$

for fixed "lags" r and s ($r > s > 0$) and $n \ge r$. Here θ is some binary operator, e.g. addition (mod m), subtraction (mod m), multiplication (mod m) or "exclusive or" (mod $m = 2^w$). We abbreviate these operators by $+, -, *$ and $\oplus$ respectively. Generators using $\oplus$ are also called "shift register" generators or "Tausworthe" generators [11, 18, 31].

If θ is $+$ or $-$ (mod m) then a theory of generalized Fibonacci generators can be based on the generating function

$$G(x) = \sum_{n=0}^{\infty} U_n x^n$$

which is given by

$$G(x) = P(x)/Q(x) \bmod m,$$

where

$$Q(x) = 1 - (x^r \theta x^s)$$

and $P(x)$ is a polynomial of degree at most $r-1$ determined by the initial values $U_0, \ldots, U_{r-1}$. For example, if $m = 2$ and the initial values are not all zero, then the sequence has maximal period $2^r - 1$ if and only if $Q(x)$ is a primitive polynomial (mod 2). Tables of such primitive polynomials are available [17, 32]. Verification is particularly simple if r is the exponent of a Mersenne prime (i.e. $2^r - 1$ is prime) because then we only need to check that

$$x = x^{2^r} \bmod (Q(x), 2)$$

which can be done by r squarings of polynomials (mod 2), involving a total of only $O(r^2)$ operations [33]. The more usual formulation in terms of r by r matrices [20, 21] instead of polynomials is less efficient computationally because matrix multiplication is more expensive than polynomial multiplication.

If $m = 2^w$ and the lags r and s are chosen correctly, it is possible to obtain period

$$p = \begin{cases} 2^r - 1 & \text{if } \theta = \oplus, \\ 2^{w-1}(2^r - 1) & \text{if } \theta = \pm \bmod m, \\ 2^{w-3}(2^r - 1) & \text{if } \theta = * \bmod m. \end{cases}$$

The initial values must be odd for $\theta = *$, not all even for $\theta = \pm$, and not all zero for $\theta = \oplus$. For precise conditions, see [5, 21]. We see one advantage of the generalized Fibonacci generators over linear congruential generators – the period can be made very large by choosing r large. However, one should refrain from using more than $2^r - 1$ numbers from such generators with $\theta = \pm$, because U_n and $U_{n+p/2^k}$ differ in at most k bits $(0 < k < w)$.

Marsaglia [20] reports the results of statistical tests on the generators $F(17, 5, \theta)$, $F(31, 13, \theta)$ and $F(55, 24, \theta)$. The results for $\theta = \oplus$ are poor – several tests are failed. All tests are passed for the generators $F(607, 273, \theta)$ and $F(1279, 418, \theta)$, so the conclusion is that $\oplus$ generators should only be used if the lag r is large.

Marsaglia's results for $\theta = \pm$ are good with one exception – the generators with $r \le 55$ fail the "Birthday Spacings" test. This is not surprising because (in the case $\theta = -$) the recurrence

$$U_n = U_{n-r} - U_{n-s} \bmod m$$

shows that a small value of U_n implies a small difference $U_{n-r} - U_{n-s}$. The Birthday Spacings test is designed to test if such small differences occur more (or less) often than they should. If $\theta = +$ a similar argument applies, since

$$U_{n-s} = U_n - U_{n-r} \bmod m.$$

The conclusion is that these generators are probably acceptable if r and s are sufficiently large (not necessarily as large as for the $\oplus$ generators).

Our argument against the simple Fibonacci recurrence also applies to the generalized Fibonacci recurrence

$$u_n = u_{n-r} \pm u_{n-s} \bmod 1,$$

because only four of the six orderings of (u_n, u_{n-r}, u_{n-s}) can occur. A statistical test based on this fact can easily "fail" any $F(r, s, \pm)$ generator. Even if r and s are not assumed to be known, the test can check all possible r and s satisfying $0 < s < r < B$ say, where B is a prescribed bound. The storage and number of operations required are of order B^2. We call such a test a "Generalized Triple" test. Clearly the existence of such tests is a reason for choosing a large r.

Marsaglia's results indicate that generalized Fibonacci generators with $\theta = * \bmod m$ are acceptable. In fact, these are the only class of generators other than "combination" generators to pass all his tests. Unfortunately, it is more difficult to implement multiplication (mod m) than addition/subtraction (mod m) because of the requirement for a double-length product unless m is small enough for m^2 to be representable in single-precision. When testing generalized Fibonacci generators with $\theta = +$ Marsaglia used small r; such generators should be acceptable for sufficiently large r.

Empirically, and with some theoretical justification, we have found two ways to improve the performance of generalized Fibonacci generators on the Birthday Spacings and Generalized Triple tests. The simplest is to include small odd integer multipliers α and β in the generalized Fibonacci recurrence, i.e.

$$U_n = \alpha U_{n-r} + \beta U_{n-s} \bmod m.$$

We denote these generators by $G(r, s, \alpha, \beta)$. The theory goes through with minor modifications. By Theorem 2 of [5], the period is $2^{w-1}(2^r - 1)$ if $m = 2^w$, provided the trinomial $x^r + x^s + 1$ is primitive (mod 2) and $U_0, \ldots, U_{r-1}$ are not all even.

An alternative which avoids difficulties with the Birthday Spacings and Generalized Triple tests but is almost as fast and easy to implement as the $F(r, s, \pm)$ generators is to include another term in the generalized Fibonacci recurrence, i.e.

$$U_n = U_{n-r} \pm U_{n-s} \pm U_{n-t} \bmod m,$$

where $r > s > t > 0$ are suitably chosen lags. We call such generators "4-term generalized Fibonacci" generators in contrast to the usual 3-term generators. For a discussion of 4-term generators, we refer to [4].

4 Comments on some available generators

We have discussed RANDU in the sub-section on linear congruential generators, and shown why similar generators are not to be recommended. Even with an improved choice of multiplier, such generators suffer from having a period $p = 2^{31}$ which is far too short. It takes less than one minute to run through the whole period of such a generator on the Fujitsu VP 2200/10 at ANU. Similar comments apply to any linear congruential generator with modulus representable as a 32-bit integer. Linear congruential generators with larger modulus (e.g. 2^{64}) are difficult to implement efficiently on many machines because they require multiple-precision integer arithmetic.

Attempts have been made to improve the statistical properties of linear congruential generators by shuffling (see [16], page 32). This does usually improve the d-dimensional uniformity of the output for $d > 1$, but does little (if anything) to increase the period. Shuffling does not improve the performance on some statistical tests, e.g. the Birthday Spacings test. Most important, shuffling is slow and difficult to vectorize, so it is inappropriate to use it on vector processors.

Instead of shuffling, the output of two generators can be combined by addition (mod 1) or $\oplus$. These operations vectorize, but the combination generator is still two to three times slower than a single generator, and the results are not guaranteed to pass all statistical tests (see comments on the "Super-Duper" generator in the next section). For these reasons, it seems preferable to use a single (good) generator.

Some generators, especially those based on the linear congruential method with multipliers a power of two, suffer from poor resolution, because they return only single-precision (32-bit) real numbers. Since most serious work on vector processors and fast parallel machines use double-precision (64-bit) real numbers, it is desirable for a library routine to return double-precision numbers (with the low-order 32 bits not all zero). If N random numbers are used in a simulation, there is not much point in requiring resolution finer than $1/N$. Thus, it may be acceptable for a small number of the low-order bits (say up to four) to be zero when 64-bit numbers are returned.

The trend appears to be for generalized Fibonacci generators to supplant linear congruential generators. For example, the recent reviews [1, 15] approve of the generators $F(r,s,\pm)$ mod 2^w, provided the lags are large. Anderson uses $F(607,273,-)$ and shows how it can be implemented efficiently on a vector processor.

Siemens Nixdorf in collaboration with the University of Karlsruhe have implemented a package of random number generators (RAND/VP). The generators are adapted from the generator UNI of [23], which is based on the generalized Fibonacci generator $F(97,33,-)$. Marsaglia [22] now prefers his VLP generators (described below).

The algorithm used in RAND/VP "has been modified to generate several streams of random numbers in parallel" [13]. Presumably this means that the recurrence

$$u_n = u_{n-97} - u_{n-33} \bmod 1$$

is applied to *vectors* of length $v > 1$ rather than to single real numbers u_n. This is equivalent to using the generator $F(97v, 33v, -)$. As far as the statistical properties are concerned, it would be better to use a single generalized Fibonacci generator $F(r,s,-)$ with lags $0 < s < r \simeq 97v$ chosen to give maximal period. As we outline below, it is possible to vectorize the generation of a single stream of random numbers.

In his recent papers [22, 24] Marsaglia recommends a new class of generators, termed "very long period" (VLP) generators. These are similar to generalized Fibonacci generators but can achieve periods close to 2^{rw}, whereas the generalized Fibonacci generators can "only" achieve period $O(2^{r+w})$. Our tests indicate that the VLP generators perform almost as badly on the Birthday Spacings test as the generalized Fibonacci generators using addition/subtraction. Perhaps this is why Marsaglia recommends combining them with a different class of generator [22]. In any event, the VLP generators require the computation of a "carry" or "borrow" which propagates to the next term in the sequence. This dependence causes a problem on vector processors. Although it is possible to vectorize each carry/borrow propagation step, the number of operations required in the inner loop is 2–3 times greater than for the generalized Fibonacci generators.

5 Suggested vector and parallel algorithms

In this subsection we consider which classes of random number generators are suitable for implementation on vector processors and/or parallel machines.

Linear congruential generators of the form

$$U_{n+1} = (aU_n + c) \bmod m$$

can be implemented efficiently on a parallel machine with k processors or a vector processor with vector registers of length k, using the relations

$$U_n = a^n U_0 + \left(\frac{a^n - 1}{a-1}\right) c \bmod m$$

and

$$U_{k(n+1)} = a^k U_{kn} + \left(\frac{a^k - 1}{a-1}\right) c \bmod m.$$

Note that $a^n \bmod m$ can be computed in $O(\log n)$ operations using the binary representation of n [16].

Despite the possibility of efficient implementation, linear congruential generators are not recommended because of their poor d-dimensional distribution (for $d > 1$) and small periods. Even when the multiplier a is chosen to pass the Spectral Test, the multiplier a^k which is effectively being used to produce each k-th term in the sequence may fail (in fact this is sure to happen for some values of k).

The statistical properties of generators can be improved by combining two or more independent generators, using $\pm$ (mod m), $\oplus$, or shuffling. If generators with relatively prime periods p_1 and p_2 are combined, the period of the combined generator is generally $p_1 p_2$.

Combination generators are not recommended because they are slow – usually two to three times slower than the component generators. Also, they are not guaranteed to have good statistical properties. For example, Marsaglia's "Super-Duper" combination generator, which combines a linear congruential generator and a shift register generator, fails the MTUPLE test on substrings of low order bits [20].

r	s	r	s
127	97	3217	2641
258	175	4423	3004
521	353	9689	5502
607	334	19937	10095
1279	861	23209	13470
2281	1252	44497	23463

Table 1: 3-term generators

Generalized Fibonacci generators can be implemented efficiently on vector/parallel machines [1]. The Tausworthe/shift register generators must be treated with suspicion because of their poor statistical properties for small and moderate lags [20]. Presumably the discrepancy between their behaviour and ideal behaviour is a decreasing function of the lag r, so they pass standard tests if r is sufficiently large, but for any fixed r they would probably fail a sufficiently stringent test. This may or may not be relevant in a particular application, but it is hardly acceptable for a library routine.

Generalized Fibonacci generators $F(r,s,*)$ based on multiplication (mod 2^w) pass all of Marsaglia's tests [20] (w is not specified, but presumably is at least 16). The problem with these generators is that they are difficult to implement without double-precision multiplication if w exceeds *half* the word-length. For example, to obtain 56-bit fractions ($w = 56$) would require the multiplication of 56-bit integers mod 2^{56}. A library routine should have close to the maximum precision allowed by the hardware, so returning a "random" number whose low-order bits are all zero is unacceptable.

Generalized Fibonacci generators $F(r,s,\pm)$ based on addition or subtraction (mod 2^w) are well-suited to vector and parallel machines. The lag r should be chosen large enough that the Birthday Spacings test is passed. Preliminary tests indicate that $r > 100$ is satisfactory. This is not a serious constraint because it is desirable to choose a large r to give long vector lengths and a long period. The second lag s should not be either too small or too close to r. We recommend $s \simeq \rho r$, where $\rho = 0.618\cdots$ is the golden ratio, subject to the constraint that the polynomial $x^r + x^s + 1$ is primitive (mod 2) so the period is at least $2^r - 1$. Some suitable pairs (r, s) are given in Table 1.

A nice feature of the generalized Fibonacci generators $F(r,s,\pm)$ is that they can be implemented in floating-point arithmetic without conversion from integer to floating-point (see [16], page 27). This may give higher speed than competing methods, and also allows the generation of full (or almost full)-precision numbers. For example, on a machine with 32-bit integer arithmetic and 56-bit floating-point fractions, it is possible to generate random numbers with all 56 bits nonzero and random. The idea is to use floating-point numbers to represent integers scaled by 2^{-56} and ensure that all floating-point additions/subtractions are exact. Despite this observation, the fastest implementation on machines with fast floating-point hardware may well be to use the hardware integer $\leftrightarrow$ real instructions to obtain the fractional parts of numbers (or vectors of numbers). This is especially likely to be true if multipliers α, β other than ± 1 are used.

At some cost in performance any worries about the Birthday Spacings test can be avoided by using a 4-term (rather than the usual 3-term) recurrence, or by using a 3-term recurrence with odd integer multipliers $\alpha, \beta > 1$ (see above). Our implementation on the Fujitsu VP 2200 indicates that the loss of performance is 26 percent for the 4-term recurrence, and 23 percent for the 3-term recurrence with non-unit multipliers.

6 Architectural considerations

In this section we consider the implementation of the generalized Fibonacci method $F(r,s,+)$ on the following classes of machines –

1. Single-processor serial machines, e.g. Sun Sparcstation, IBM PC, ...
2. Vector processors, e.g. Fujitsu VP series, Cray.
3. Local-memory MIMD multiprocessors, e.g. Intel Delta, Fujitsu AP 1000.
4. Shared-memory SIMD multiprocessors, e.g. Connection Machine (CM 2).

On single-processor serial "Von Neumann" machines there is little justification for using the larger values of r given in Table 1. Probably $r \le 1279$ is sufficient. Large values of r require correspondingly large arrays, which may be a problem on machines such as IBM PCs. Performance for large r may also be degraded because of an increase in cache misses.

On a vector processor we can assume that ample memory is available, even for the largest r given in Table 1. We certainly should use r large enough that vector operations on vectors of length s and $r - s$ can be performed efficiently. Comments based on our experience in implementing the generalized Fibonacci methods on the Fujitsu VP 2200 are given in Section 7.

On a local-memory MIMD multiprocessor such as the Fujitsu AP 1000, the comments regarding implementation on single-processor machines apply, because each processor is such a machine. (If each processor has a vector unit and sufficient memory, as on the CM 5, then the comments on vector processors apply.) Our initialization scheme (Section 8) ensures that different random sequences will be generated in each processor *provided each processor uses a different seed.* Thus, it might be wise for the initialization routine to append the processor ID to the user-supplied seed. For example, on the AP 1000, with a maximum of 1024 processors (also called "cells"), the seed on processor `cid` might be `1024*useed + cid`, where `useed` is the seed supplied by the user. We assume here that ($0 <$ `useed` $< 2^{21}$).

On a shared-memory SIMD multiprocessor such as the Connection Machine (CM 1 or CM 2), with a large number (say P) of relatively slow processors, it is not appropriate for each processor to generate an independent segment of the sequence (u_n). This would require total memory of order Pr words. It is better to group the processors in sets of some moderate size (say 256 for the sake of example). The processors in each set can cooperatively generate a single segment of the sequence (u_n), with each processor generating each 256-th number in the sequence. This scheme can be implemented efficiently provided $s \geq 256$, and the total memory requirement is only of order $Pr/256$ words. We can think of each set of 256 processors as emulating a single vector processor, and operating on vectors of length 256.

7 Vectorization

In this section we consider in more detail the implementation of a generalized Fibonacci generator on a vector processor, using our experience in implementing such a generator on the Fujitsu VP 2200.

The usual implementation of generalized Fibonacci generators involves a ring buffer of length at least r (the larger lag). This is inconvenient on a vector processor, but can be avoided in several ways. One way is described in Anderson's review [1]. We have implemented another idea which is more efficient because it involves less copying. We have also removed Anderson's restriction on the number of random numbers which may be returned on each call of the library routine.

Although it is clear in principle that the inner loops of the implementation should vectorize, some care has to be taken to avoid dependencies which would inhibit vectorization. For example, the recurrence

$$u_n = u_{n-r} + u_{n-s} \bmod 1$$

can be vectorized without difficulty in a loop whose repeat count is at most $\min(r, s)$. In our implementation we have found that it is convenient to assume that $r - s < s < r$ (which is not a serious constraint, since we can always replace s by $r - s$). For good vector performance, $\min(s, r - s)$ should not be too small.

In our implementation the user asks for any positive number (say N) of random numbers, and provides a buffer `BUF` of size at least N words. We also maintain an array `WORK` of size r, not accessed directly by the user, and the index of the last word used in `WORK`. There are essentially two cases –

1. $N < 2r$. Copy previously-generated numbers from `WORK` to `BUF`. Generate batches of r numbers directly in `WORK` as required. Each batch can be generated with two vectorizable loops (generating s and $r - s$ numbers).

2. $N \geq 2r$. To avoid the copying overhead implicit in case 1, generate numbers directly in `BUF`. For the first r numbers this requires access to `WORK`, but once the first r numbers are in `BUF`, batches of s numbers can be generated using a vectorizable loop. The last r numbers must be copied into `WORK` for use on the next call, as it is not assumed that `BUF` is preserved between calls.

Logically, case 2 is superfluous, but it is important if the best performance is to be obtained on machines for which arithmetic is as fast as (or faster than) memory access.

There is a question as to the best way to code the "mod 1" operation in Fortran or C. For example, in Fortran we could use

```
X(N) = X(N) - INT(X(N))
```

if it is known that `X(N)` ≥ 0, or

```
IF (X(N) .GE. 1D0) X(N) = X(N) - 1D0
```

if it is known that $0 \leq$ `X(N)` < 2. On the VP 2200 both forms vectorize, but the first is faster because it avoids the need for masked operations. It is also more general, because if applies even if `X(N)` ≥ 2 (which may be the case if $\alpha > 1$ or $\beta > 1$). Note that both forms need modification if $\alpha < 0$ or $\beta < 0$ – in this case we would have to use something like

```
X(N) = X(N) + BIAS - INT(X(N) + BIAS)
```

where `BIAS` $\geq \max(|\alpha|, |\beta|, |\alpha + \beta|)$, or possibly

```
IF (X(N) .LT. 0D0) X(N) = X(N) + 1D0
```

if $(\alpha, \beta) = (1, -1)$ or $(-1, 1)$.

Care has to be taken in the initialization and choice of w because of the possible loss of the least significant bit(s) when an intermediate result is greater than 1. (It is possible to avoid losing any bits, but only with a significant performance penalty.) With the moderate restriction

$$|\alpha| + |\beta| \leq 16,$$

α	Terms	Pipes	Loops	Cycles
$\alpha > 1$	3	1	normal	3.32
$\alpha > 1$	3	2	normal	2.71
$\alpha > 1$	3	2	unrolled	2.71
$\alpha = 1$	3	2	normal	2.31
$\alpha = 1$	3	2	unrolled	2.21
$\alpha = 1$	4	2	unrolled	2.78

Table 2: Cycles/number on the VP 2200/10

a straightforward implementation loses only 4 bits (one hex digit), which seems acceptable. In order to preserve the theory, which assumes exact arithmetic, the initialization routine should ensure that the bits which could be lost due to rounding errors are always zero. For example, on a machine with a 56-bit fraction and $|\alpha|+|\beta| \le 16$, only the leading 52 bits in the fraction should be set by the initialization routine. This amounts to using $w = 52$ rather than $w = 56$.

The results of some experiments on the Fujitsu VP 2200/10 at ANU are summarised in Table 2. In order to understand their significance, it is necessary to know that the VP 2200/10 has two load/store pipes which communicate between memory and the vector registers. Each pipe can load or store one 64-bit word per cycle. A cycle is 3.2 nsec. There is a vector unit with two multiply/add pipelines, so potentially two multiplies and two adds can be performed each cycle. In Table 2, "cycles" is the number of clock cycles per random number generated, using large lags r and s. The results show that we save about 0.6 cycles per number by persuading the compiler to use both load/store pipes (otherwise memory bandwidth is halved) and another 0.4 cycles if one of the multipliers α, β is 1. Performance is no better if $\alpha = \beta = 1$ than if $\alpha = 1$, $\beta > 1$ because one multiplication instruction is "free", using a vector instruction which performs a multiplication and addition. Loop unrolling saves another 0.1 cycles if $\alpha = 1$ but not if $\alpha > 1$, $\beta > 1$.

At 2.21 cycles per number we can generate 141 million numbers per second on the Fujitsu VP 2200/10. This is about 2.6 times faster than a vectorized implementation of the linear congruential method (with unacceptably small period 2^{31}) and 34 times faster than a method which shuffles the output of a linear congruential generator but does not vectorize.

The number of cycles per number for the 4-term recurrence method is about 0.6 more than for the corresponding 3-term recurrence method. In this case loop unrolling is worthwhile and up to two multiplications are "free".

The computations required during initialization (described in the next section) can be vectorized without difficulty.

8 Initialization

For both the 3-term and 4-term generalized Fibonacci methods, an important aspect is the initialization of $U_0, \ldots, U_{r-1}$. This is often done using another generator, e.g. a linear conguential generator. However, this introduces a source of confusion, loss of portability, and possible statistical problems. It seems better to avoid the use of any other generator. We outline how this may be done, and how the requirements given in Section 1 can be satisfied.

The idea is that the user will provide a single-precision integer seed, and the initialization will *guarantee* that any two different seeds will give non-overlapping (sub-)sequences for all practical purposes. On a parallel machine, different processors need only ensure that they use different seeds. The key point is that the least significant bits satisfy a recurrence mod 2, and by polynomial squaring we can efficiently "skip" along the sequence of least significant bits. If the least significant bits in two subsequences differ, then because of carries the higher bits are also sure to differ. Our implementation of the 3-term method has $r > 90$ and skips past $seed \times 2^{60}$ elements of the sequence, so subsequences of $2^{60} > 10^{18}$ elements are guaranteed to be different so long as they are initiated with different seeds.

To be more specific, consider the three-term generator

$$U_n = \alpha U_{n-r} + \beta U_{n-s} \bmod 2^w, \qquad (1)$$

where α and β are odd. (In practice, we work with $u_n = U_n/2^w$, but this makes no essential difference.) The least-significant bits $x_n = U_n \bmod 2$ satisfy the recurrence

$$x_n = x_{n-r} + x_{n-s} \bmod 2. \qquad (2)$$

We assume that at least one of $U_0, \ldots, U_{r-1}$ is odd, so at least one of $x_0, \ldots, x_{r-1}$ is nonzero. For simplicity we may as well assume that

$$x_0 = 1, x_1 = \cdots = x_{r-1} = 0.$$

Let

$$Q(t) = t^r - t^{r-s} - 1.$$

From the theory [5, 11] of linear recurrences (mod 2), if

$$t^n = \sum_{j=0}^{r-1} a_{n,j} t^j \qquad \bmod (2, Q(t)),$$

then

$$x_n = \sum_{j=0}^{r-1} a_{n,j} x_j \qquad \bmod 2.$$

Suppose we want to "skip" to x_n, where n is large, so we do not want to generate all of the intervening sequence. We can compute $t^n \bmod (2, Q(t))$ in $O(\log n)$ steps using the "binary method" (see [16], Sec. 4.6.3). Arithmetic on the coefficients is always performed mod 2, and at each step a reduction mod $Q(t)$ is performed. Because $Q(t)$ is a trinomial, squaring (mod 2) and reduction take only $O(r)$ operations, so x_n can be computed with $O(r \log n)$ operations. The working space required is only about $2r$ bits.

To start generating the sequence from index n we need not only x_n but also $x_{n+1}, \ldots, x_{n+r-1}$. To obtain these we may compute $t^n, t^{n+1}, \ldots, t^{n+r-1} \bmod (2, Q(t))$ or, perhaps simpler, compute $x_r, \ldots, x_{2r-2}$ from the recurrence (2) and use

$$x_{n+k} = \sum_{j=0}^{r-1} a_{n,j} x_{j+k} \qquad \bmod 2$$

for $k = 0, \ldots, r-1$.

Assuming, as above, that no more than 2^{60} consecutive random numbers will be required, we take

$$n = 2^{60} seed$$

and generate $x_n, \ldots, x_{n+r-1}$ as described. Now use these values as starting values for a sequence satisfying the recurrence (1). Because the last bits differ (for at least 2^{60} terms) from those obtained from any other seed, the "disjoint subsequence" requirement of Section 1 is satisfied.

In practice it is unsatisfactory to use the first few numbers generated in this way, because only their low order bits are nonzero (recall that $u_n = U_n/2^w$). We need to generate $O(rw)$ numbers, using the recurrence (1), and discard them. The following adaptive scheme takes a negligible amount of time and appears to be satisfactory: generate batches of r numbers (say $v_0, \ldots, v_{r-1}$) until $10v_0 > 1$ and $10r \min(v_0, \ldots, v_{r-1}) > 1$; then generate and discard 10 more batches of r numbers.

9 Conclusion

We have considered the requirements for uniform pseudo-random number generators on modern vector and parallel machines, and considered the advantages and disadvantages of various popular classes of methods, including linear congruential and generalized Fibonacci. We have argued that generalized Fibonacci generators (with a suitable choice of parameters) have good statistical properties and can be implemented efficiently on vector processors and parallel machines. A good scheme for the initialization of these generators has been outlined, and the results of an implementation on a Fujitsu VP 2200/10 vector processor have been described. Our implementation appears to satisfy the requirements of uniformity, independence, long period, repeatability, portability, disjoint subsequences for different seeds, and efficiency.

Acknowledgements

Thanks are due to Dr A. Cleary, Dr R. Gingold, Dr M. Hegland and Dr P. Price for their assistance. The ANU Supercomputer Facility provided time on the VP 2200/10 for the development and testing of our implementation. This work was supported in part by a Fujitsu-ANU research agreement.

References

[1] S. L. Anderson, "Random number generators on vector supercomputers and other advanced architectures", *SIAM Review* 32 (1990), 221-251.

[2] R. P. Brent, "Algorithm 488: A Gaussian pseudo-random number generator (G5)", *Communications of the ACM* 17 (1974), 704-706.

[3] R. P. Brent (editor), *CAP Workshop 1991 – Proceedings of the Second Fujitsu-ANU CAP Workshop*, Australian National University, Canberra, November 1991.

[4] R. P. Brent, *Uniform Random Number Generators for Vector and Parallel Computers,* Technical Report TR-CS-92-02, Computer Sciences Laboratory, ANU, March 1992.

[5] R. P. Brent, *On the Periods of Generalized Fibonacci Recurrences,* Technical Report TR-CS-92-03, Computer Sciences Laboratory, ANU, March 1992.

[6] H. S. Bright and R. L. Enison, "Quasi-random number sequences from a long-period TLP generator with remarks on application to cryptography", *Computing Surveys* 11 (1979), 357-370.

[7] P. L'Ecuyer, "Efficient and portable combined random number generators", *Communications of the ACM* 31 (1988), 742.

[8] G. S. Fishman and L. R. Moore, "An exhaustive analysis of multiplicative conguential random number generators with modulus $2^{31} - 1$ ", *SIAM J. Sci. Stat. Computing* 7 (1986), 24-45.

[9] P. Frederickson, R. Hiromoto and J. Larson, "A parallel Monte Carlo transport algorithm using a pseudo-random tree to guarantee reproducibility", *Parallel Computing* 4 (1987), 281-290.

[10] M. Fushimi and S. Tezuka, "The k-distribution of generalized feedback shift-register pseudorandom numbers", *Communications of the ACM* 26 (1983), 516-523.

[11] S. W. Golomb, *Shift Register Sequences*, Holden-Day, San Francisco, 1967, Sections 2.5 and 3.4.

[12] B. F. Green, J. E. K. Smith and L. Klem, "Empirical tests of an additive random number generator", *J. ACM* 6 (1959), 527-537.

[13] O. Haan, *RAND/VP Users Guide*, Siemens Nixdorf, Munich, 1992.

[14] D. W. Heermann and A. N. Burkitt, "Parallelization of the Ising model and its performance evaluation", *Parallel Computing* 13 (1990), 345-357.

[15] F. James, "A review of pseudorandom number generators", *Computer Physics Communications* 60 (1990), 329-344.

[16] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (second edition), Addison-Wesley, Menlo Park, 1981.

[17] Y. Kurita and M. Matsumoto. Primitive t-nomials ($t = 3, 5$) over $GF(2)$ whose degree is a Mersenne exponent ≤ 44497. *Mathematics of Computation*, 56:817-821, 1991.

[18] T. G. Lewis and W. H. Payne, "Generalized feedback shift register pseudorandom number algorithm", *J. of the ACM* 20 (1973), 456-468.

[19] G. Marsaglia, "Random numbers fall mainly on the planes", *Proc. Nat. Acad. Sci. USA* 61, 1 (1968), 25-28.

[20] G. Marsaglia, "A current view of random number generators", *Computer Science and Statistics: The Interface* (edited by L. Billard), Elsevier Science Publishers B. V. (North-Holland), 1985, 3-10.

[21] G. Marsaglia and L. H. Tsay, "Matrices and the structure of random number sequences", *Linear Algebra and Applications* 67 (1985) 147-156.

[22] G. Marsaglia, B. Narasimhan, A. Zarif, "A random number generator for PC's", *Computer Physics Communications* 60 (1990), 345-349.

[23] G. Marsaglia, A. Zaman and W. W. Tsang, "Toward a universal random number generator", *Statist. Probab. Lett.* 9 (1990), 35-39.

[24] G. Marsaglia and A. Zaman, "A new class of random number generators", *The Annals of Applied Probability* 1 (1991), 462-480.

[25] A. De Mattheis and S. Pagnutti, "A class of parallel random number generators", *Parallel Computing* 13 (1990), 193-198.

[26] J. von Neumann, "Various techniques used in connection with random digits", *The Monte Carlo Method*, National Bureau of Standards (USA) Applied Mathematics Series 12 (1951), 36.

[27] S. K. Park and K. W. Miller, "Random number generators: good ones are hard to find", *Communications of the ACM* 31 (1988) 1192-1201.

[28] G. E. Percus and M. H. Kalos, "Random number generators for MIMD parallel processors", *J. Parallel and Distributed Computing* 6 (1989), 477-497.

[29] W. P. Petersen, "Some vectorized random number generators for uniform, normal, and Poisson distributions for CRAY X-MP", *J. Supercomputing* 1 (1988), 327-335.

[30] J. F. Reiser, *Analysis of Additive Random Number Generators*, Ph. D. thesis and Technical Report STAN-CS-77-601, Stanford University, 1977.

[31] R. C. Tausworthe, "Random numbers generated by linear recurrence modulo two", *Mathematics of Computation* 19 (1965), 201-209.

[32] N. Zierler and J. Brillhart, "On primitive trinomials (mod 2)", *Information and Control* 13 (1968), 541-554. Part II, *ibid* 14 (1969), 566-569.

[33] N. Zierler, "Primitive trinomials whose degree is a Mersenne exponent", *Information and Control* 15 (1969), 67-69.