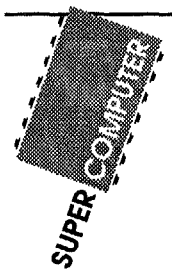
Pseudorandom number generation on supercomputers

Kleijnen, J.P.C.

Published in:
Supercomputer

Publication date:
1989



Contributions

Pseudorandom number generation on supercomputers

Jack Kleijnen

Department of Information Systems and Auditing, School of Business and Economics, Catholic University Brabant, P.O. Box 90153, 5000 LE Tilburg, The Netherlands

© *Supercomputer* 34, VI–6
Received June 1989

Pseudorandom number generators are essential in Monte Carlo simulation. Supercomputers should generate these numbers in parallel. Several procedures are evaluated, and one practical procedure is developed further.

Random numbers are the basic elements of stochastic simulation and Monte Carlo models. Examples of such models are simulations of queuing networks and Monte Carlo studies of statistical estimators. A computer does not use truly random numbers. Instead the computer generates *pseudorandom* numbers; that is, a deterministic algorithm generates outputs that behave as if these outputs were random numbers. We concentrate on one class of algorithms, namely linear congruential generators. These generators (which are treated below) are popular in management science, mathematical statistics, computer science, and many more scientific disciplines; see [1].

The problem is that, by definition, pseudorandom generators are suspect; that is, deterministic algorithms are surmised to produce non-random outputs. So a particular generator is used only until a new generator is developed that more closely simulates a truly random number sequence; see [2–4]. Moreover, when new generations of computers are introduced, new generators must be developed. For example, 8-bit personal computers could not efficiently use algorithms developed for 32-bit machines. Supercomputers still employ mostly generators that were developed for scalar machines, but this practice is very inefficient, as we shall see.

Pseudorandom number generators

Control Data Corporation (CDC) offers the Cyber 205, which runs Fortran 200; see [5]. Since Fortran 200 is a superset of standard Fortran, standard algorithms can be utilized, but they do not take advantage of the vector facilities. Scalar computers often employ linear congruential generators:

$$x_{j+1} = (ax_j + c) \; mod \; m \; (j = 0, 1, 2, \ldots), \qquad (1)$$

where the multiplier a, the constant c, the modulus m, and the *seed* x_o are integers. When c is zero, the generator is called multiplicative congruential. Obviously $r_j = x_j/m$ yields $0 \leq r_j < 1$. An efficient algorithm results if we take $m = 2^w$ where w depends on the computer's

word size; for example, Fortran 200 uses $m = 2^{47}$, but the IMSL library (developed for scalar computers) uses $m = 2^{31} - 1$. However, there are other considerations besides efficiency.

These generators should yield results r_j that are statistically independent; that is, the observed sequence $r_0, r_1, \ldots, r_n$ should not provide any information about the next sequence $r_{n+1}, r_{n+2}, \ldots$. It turns out to be extremely difficult to meet this requirement, as many authors show; see [1–4]. It is possible to derive necessary conditions which, however, are not sufficient. For example, because of computer efficiency we may select a modulus $m = 2^w$ and a constant $c = 0$; then a multiplier $a = 4g \pm 1$ with odd integer g yields the maximum cycle length or *period* (say) $h = m/4 = 2^{w-2}$ (that is, if the generator starts with seed x_0 then $x_h = x_0$ and hence $x_{h+1} = x_1$, and so on). A long period, however, is not sufficient; so statistical tests must be applied to the empirical results $(r_0, r_1, \ldots)$ to check if several types of statistical dependence are absent indeed. For example, two-tuples (r_0, r_1), (r_2, r_3), $(r_4, r_5), \ldots$ should be uniformly distributed over the unit square.

These considerations explain why practitioners do not choose their own parameters a, c and m; instead they rely on well-tested generators. An example is the IMSL routine that uses $a = 16807$, $c = 0$, and $m = 2^{31} - 1$. A popular generator is provided by the NAG library with $a = 13^{13}$, $c = 0$, and $m = 2^{59}$ (64-bit words). Textbooks on simulation discuss other parameter combinations. We now return to the Cyber 205.

Generators on the Cyber 205

Fortran 200 provides a scalar function called RANF, which uses the multiplicative congruential generator with $m = 2^{47}$ and $a = 84000335758957$. Alternatively, to generate a vector of pseudorandom numbers, Fortran 200 provides the vectorized subroutine VRANF. Unfortunately, the documentation on this subroutine is very meager, so we do not know exactly how VRANF operates.

Suppose that a given simulation experiment requires N pseudorandom numbers in total, for example, $N = 1,000,000$; we need not know the exact value of N in practice. Number theory supplies the period H of a given generator. The generators that are used in practice, have a relatively long period: $h \gg N$.

On a vector computer, results should be computed in parallel. This means that the N pseudorandom numbers should not be computed recursively (unfortunately, as (1) shows the linear congruential generator is recursive). Instead (say) J numbers should be produced in parallel. The literature and manuals suggest that J be at least 50; otherwise the vector architecture is inefficient and it is better to use the computer in scalar mode. There is also an upper limit $J \leq 65,535$ one the Cyber 205 because 16 bits are used for addressing; see [6, p. 26]. Hence a simulation experiment requiring N numbers must call the parallel routine $\lceil N/J \rceil$ times where we use $\lceil\ \rceil$ for "rounding upwards to the next integer". We

may imagine an $I \times J$ matrix of pseudorandom numbers, where J numbers are generated in parallel and I calls are made to that vector routine. We shall survey different solutions to this problem.

Different multipliers

We can generate J pseudorandom numbers in parallel, if we use J multipliers and constants in the linear congruential relationship; that is (1) becomes

$$x_{i+1,j} = (a_j x_{ij} + c_j) \; mod \; m \; (j = 1, \ldots, J) \; (i = 1, 2, \ldots).$$

So the vector of old numbers $\tilde{x}_1 = (x_{11}, x_{12}, \ldots, x_{1J})^T$ yields the vector of new numbers $\tilde{x}_2 = (x_{21}, x_{22}, \ldots, x_{2J})^T$ or in Fortran 200:

```
XNEW(1:J) = A(1:J) * XOLD(1:J) + C(1:J)    (2)
```

where `X(1:J)` denotes a vector `X` with `J` elements that starts at address 1; see [6]. After the modulus operation, realized in vector mode by the `VMOD` function, we put

```
XOLD(1:J) = XNEW(1:J)                      (3)
```

We emphasize that the elements *within* $\tilde{x}_2$ or `XNEW` are computed in parallel.

Unfortunately, it is a problem to find as many as J multipliers a_j and constants c_j. We have seen that necessary conditions for the parameter a (and c) have been derived. These conditions are so weak that, for example, we can choose from roughly one million multipliers for $m = 2^{23}$ (half precision on Cyber 205), since the following parameters meet the conditions listed in Knuth [7]:

$$a = 2,901 + 8k_1 \text{ with } k_1 = 0, 1, 2, \ldots, 1047851 \qquad (4)$$

and

$$c = 1,775,001 + 2k_2 \text{ with } k_2 = 0, 1, 2, \ldots, 2499. \qquad (5)$$

An Mey [8] proposes sampling a and c from (4) and (5). We would add that these values should be sampled without replacement (see the next section). For $m = 2^{31} - 1$ (a prime number) there are more than 534 million multipliers that yield a full period $h = m - 1$; see [1, pp. 1194, 1197]. Unfortunately, eqs. (4) and (5) give necessary but not sufficient conditions; so the statistical behavior of a generator with random parameters is very suspect! Therefore we prefer to stick to well tested parameters; that is, we prefer existing generators implemented under IMSL, NAG, and so on.

We shall limit the next discussion to multiplicative generators ($c_j = 0$), but our discussion can be extended straightforwardly to $c_j > 0$.

A vector of seeds

A simple solution is to sample a vector of J seeds; these J seeds are sampled in scalar mode using, for example, RANF in Fortran 200. Storing those seeds in XOLD means that statements (2) and (3) become

```
XNEW (1;J) = A * XOLD (1;J)
XOLD (1;J) = XNEW (1;J)                    (6)
```

Unfortunately, such sampling may result in (say) a second seed identical to (say) the third value generated from the first seed: $x_{31} = x_{12}$. Such an event means that parts of the "matrix" of numbers are identical ($x_{31} = x_{12}$ implies $x_{41} = x_{22}, \ldots, x_{I1} = x_{(I-2)2}$), and this violates the statistical independence assumption; this assumption is made for the generator and is used in the simulation model.

Frederickson et al. [9] launched a different idea, namely sample the seeds through a second generator:

$$y_{j+1} = (by_j + d) \; mod \; m (j = 0, 1, 2, \ldots).$$

The latter generator is used to sample seeds for the generator of eq. (1). Now there are five parameters (a, c, b, d, m) to be selected. Unfortunately, correlations among pseudorandom numbers remain; see [10]. Moreover, this approach has been analyzed for specific generators only; the approach does not cover a generator with (say) $m = 2^{31} - 1$, recommended in [2]; see [1] for other recommended parameters. This criticism leads to the following idea.

A vector of seeds spaced 100,000 apart

Fishman [2] proves that, given a seed x_0 and I calls to the scalar generator (see eq. (1) with $c = 0$), the resulting number x can be derived without knowing the intermediate numbers $(x_1, x_2, \ldots, x_{I-1})$:

$$x_I = (a^I x_0) \; mod \; m.$$

So if we want to generate J numbers in parallel (such that $I \times J \geq N$), then we can start with the following vector of seeds:

$$\tilde{x}_1 = (x_0, (a^I x_0) \; mod \; m, (a^{2I} x_0) \; mod \; m, \ldots, (a^{(j-1)I} x_0) \; mod \; m, \ldots, (a^{(J-1)I} x_0) \; mod \; m)^{\mathrm{T}}. \quad (7)$$

Unfortunately, we cannot implement this mathematical solution straightforwardly, since overflow occurs when computing $a^{(j-1)I}$. The overflow problem in pseudorandom number computation is also discussed in [1, p. 1195].

Fishman [2, pp. 481–487] also tabulates 400 seeds spaced 100,000 apart, for three different scalar multiplicative congruential generators; also see

[3]. He provides these three tables (each with 400 seeds) to decrease the variance of simulation responses.

We can use Fishman's tabulated values for parallel generation of pseudo-random numbers. His seeds s_j $(j = 1, \ldots, 400)$ imply $s_2 = x_{100,000} = (a^{100,000} s_1)\ mod\ m$ and $s_3 = x_{200,000} = (a^{100,000} s_2)\ mod\ m$, and so on; also see eq. (7). So suppose we use an initial vector with these 400 seeds: XOLD $= (s_1, s_2, \ldots, s_{400})^T$; also see eq. (6). Only after calling the vectorized pseudorandom subroutine 100,000 times, we shall return to the initial vector. In other words, if the total number of pseudorandom numbers N is smaller than (400×100,000 =) 40 million, this approach yields 400 different numbers in parallel. Now we discuss some practical issues and extensions.

Practical issues

Fishman gives tables only for three specific generators, namely SIMSCRIPT II, SIMPL/1-LLRANDOM, and $m = 2^{31} - 1$ with $a = 397204094$. The user may prefer a different generator, for example, NAG's subroutine or CDC's $m = 2^{47}$. We shall solve this problem below.

Even if one of these three generators is desired, keypunching 400 numbers, each consisting of up to 10 digits, is a slow and error-prone process. Therefore we propose sampling one seed from Fishman's table, and having the computer generate the remaining 399 seeds. So the computer uses eq. (1) initialized with this particular seed, and after 100,000 calls, the computer stores $s_2 = x_{100,000} = (a \times x_{99,999})\ mod\ m$, given specific parameters a and m. In total the computer must generate 399×100,000 pseudorandom numbers! Fortunately, the computer has to do this job only once: all future simulation experiments can use the internally stored table with 400 seeds. If a particular experiment requires vectors with $J < 400$ elements, then that experiment uses only the first J seeds.

If the computer generates the seeds to be stored in the initial vector, *we are no longer limited to Fishman's three tables!* We can take any generator we like; for example, we can take the generator used in experiments run on a scalar computer; these experiments are needed anyway to debug and verify the program to be run on the supercomputer; see [6, p. 5].

Moreover we are no longer limited to a vector length of 400. The longer the vector is, the more efficient the supercomputer works. So on the 205 we might be tempted to take the maximum vector length, namely $J = 65,535$, and have the computer generate all h different numbers x_j (with $j = 0, 1, \ldots, h$ and known period h); the computer then stores 65,535 numbers, namely $s_1, s_2, s_3, \ldots, s_{65535}$ with $s_2 = x_{I+1}$ and $I = \lceil h/65,535 \rceil - 1$, $s_3 = x_{2I+1}$, and so on (so nearly the whole cycle would have to be executed, namely from x_0 through $x_{65,534I}$). Elsewhere [11] we prove that this idea is statistically unacceptable. For the time being we recommend a vector length $J = 400$ (with seeds spaced 100,000 apart), which is a compromise between computer efficiency and statistical behavior.

At the end of a simulation session the user should store the last vector of pseudorandom numbers $\tilde{x}$ or $\tilde{r} = \tilde{x}/m$; *all* digits need to be saved; see [12, p. 16]. To continue this particular simulation run, the user proceeds from the vector saved at the end of the previous session. If the user wants to execute an unrelated simulation experiment, he or she can take either the last vector or the initial vector provided by the computer center. So on scalar computers the user needs to store a single pseudorandom number; on supercomputers a whole vector must be saved.

Efficiency improves if generators are developed that take advantage of the wordsize m of the particular supercomputer; that is, for the Cyber 205 $m = 2^{47}$ is efficient. The cycle length h increases as the modulus m increases. For the new modulus, a new value for the multiplier a needs to be found, applying number theory and mathematical statistics; for the Cyber 205 $a = 84000335758957$ is recommended. Finally, a new vector of seeds is to be generated for the user community.

Conclusions

Pseudorandom number generation creates problems that require the joint efforts of computer scientists for efficient implementation, number theorists for necessary conditions for the generator's parameters, and statisticians for ex-post empirical tests. On supercomputers, generators should be vectorized such that parallel computation becomes possible. The choice of the generator's parameters is crucial; that is, the sampling of different multipliers yields unacceptable statistical behavior. So the user wishes to stick to well-tested parameter values, which are available in statistical packages and simulation languages. Sampling a vector of seeds may result in dependent vectors of pseudorandom numbers. A practical solution is that the computer center generates 400 seeds 100,000 apart, and makes the resulting vector with 400 elements available to its users.

Acknowledgements

This research was sponsored by the Supercomputer Visiting Scientist Program at Rutgers University, The State University of New Jersey, during July 1988. I thank Dr. Nabil Adam at Rutgers University, Graduate School of Management, for many fruitful discussions.

References

1
Park, S. and K. Miller, *Random number generators: good ones are hard to find*, C. ACM 31, 1192–1201, 1988.

2
Fishman, G., *Principles of Discrete Event Simulation*, Wiley-Interscience, New York, 1978.

3
Bratley, P., B. Fox and L. Schrage, *A Guide to Simulation*, Springer-Verlag, New York, 1983.

4
Ripley, B., *Stochastic Simulation*, John Wiley & Sons, New York, 1987.

5
CDC, *Fortran 200 Version 1 reference manual.* Publication **60480200**, Control Data Corporation, Sunnyvale, CA 94088-3492, December 1986.

6
SARA, *Cyber 205 User's Guide; Part 3, Optimization of Fortran Programs*, SARA, Amsterdam, November 1984.

7
Knuth, D., *The Art of Computer Programming, II*, Addison-Wesley, Reading, MA, 1981.

8
An Mey, D., *Erste Erfahrungen bei der Vektorisierung numerischer Verfahren* (First experiences when vectorizing numerical procedures.) Computer Center, Technical University, Aachen, Germany, July 1983.

9
Frederickson, P., R. Hiromoto, T. Jordan, B. Smith and T. Warnock, *Pseudorandom trees in Monte Carlo*, Parallel Computing **1**, 175–180, 1984.

10
Bowman, K. and M. Robinson, *Studies of random number generators for parallel processing*, in: Heath, M. (ed.), Hypercube Multiprocessor 1987, SIAM, Philadelphia, 445–453, 1987.

11
Kleijnen, J. and B. Annink, *Pseudorandom number generators revisited*, Katholicke Universiteit Brabant, May 1989.

12
Kleijnen, J., *Selecting random number seeds in practice*, Simulation **47**, 15–17, 1986.