JOURNAL OF COMPUTATIONAL PHYSICS 81, 230–235 (1989)

Note

Parallel Processing of Random Number Generation for Monte Carlo Turbulence Simulation

1. Introduction

Certain stochastic processes in plasma physics can be simulated using random numbers with a known distribution as initial conditions for the time integration of dynamics. These integrations or *realizations* can usually be grouped into independent packets—thus rendering the problem ideally suited for multiprocessing computers. However, once these packets of realizations (we refer to each packet as a process) are sent to the individual processors, we must face a new problem—that of generating independent strings of random numbers to be used as initial conditions for the realizations in each process. We give a simple method for obtaining independent strings of random numbers for use in such Monte Carlo calculations. The method has the benefit of reproducing the identical string of numbers one would obtain on a uniprocessing computer, given that it is known in advance how the realizations are to be divided among the processes. The same procedure can be applied if this is not known in advance, as long as one can provide an upper bound on the number of random numbers required per process. In general, this upper limit can be very large. We use 3^{20} as an example, which is of order three billion.

2. The Physical Problem

In one application, we are interested in solutions to the following nonlinear equation which has been used as a model for drift-wave turbulence,

$$\left[\frac{d}{dt} + \nu_k + i\omega_k\right] \Phi_k(t) = \frac{1}{2} \sum_{k=p+q} M_{k|p,q}\, \Phi_p(t)\, \Phi_q(t), \tag{1}$$

where $\Phi_k(t)$ is a complex-valued function representing the field quantity, $(\nu_k + i\omega_k)$ is a linear growth, damping, or wave propagation term, and the right-hand side gives nonlinear coupling between different Fourier modes. We simulate the random flow field governed by Eq. (1) by integrating over an ensemble of Gaussian initial conditions.

For a typical problem, we desire the results of ≈ 4000 realizations. In one study [1], these integrations were performed on a Cray-2 computer. The realizations

0021-9991/89 $3.00

were split into four groups (1000/processor). As each integration in the ensemble finished, the results were tallied in a common block to be used in calculating average quantities such as the time-lag correlation function and the response function. For this particular problem, we were integrating three distinct complex-valued modes, and thus required six Gaussian numbers per realization. The Gaussian numbers were generated by summing uniformly distributed numbers, usually 12 per Gaussian. Therefore each processor required $\approx 72{,}000$ uniform random numbers. It is important that the numbers given to each processor be disjoint sets, since otherwise the realizations would be duplicates. Additionally, it would be useful if the sets were reproducible from run to run. Finally, it would be nice if the generated sets of numbers were the same as those one would obtain using the same random number generator on computers with different numbers of processors (e.g., when switching back and forth between our Cray-2 and Cray X/MP) without changing the (Fortran) coding. In the next section we give a means for generating sets of uniform random numbers with these characteristics.

3. Random Number Generation Method

A majority of random number generators in use today are based on linear, congruential, pseudo-random sequences of the form

$$X_{n+1} = (aX_n + c) \bmod m, \qquad n \geqslant 0, \tag{2}$$

where m is the modulus, a is the multiplier, c is the increment, and X_0 is the starting value or seed [2]. (Often, $m = 2^q$ with $q \in$ integers on binary machines. On our Cray computers, the system random number generator RANF has $m = 2^{48}$ and $c = 0$. The sequence generated has a period of 2^{46}—in other words, it samples half of the entire set of odd numbers available on a machine with a mantissa size of 2^{48}.) Other more complicated schemes generally are based on the linear congruential generator and involve, for instance, taking the terms generated in a shuffled order, or involve higher order (e.g., quadratic) difference equations of the same form.

In order to generate independent strings of numbers for multiprocessors, one normally thinks of dividing the generation process into two parts. The first part (known as the left method) is the process used to generate the seeds for each process, and the second part (the right method) generates a string of numbers using the given seeds [3, 4]. In the paper by Frederickson *et al.* [3], two sets of constants for the linear difference equation are given. One of these sets is used to produce the seeds, while the other is used to generate the random numbers for the given processor. For carefully choosen combinations of constants, a, c, and X_0, they prove that up to a specified length, the sequences generated are disjoint sets. However, Bowman and Robinson [5] point out that this method essentially takes a random number generator with m a power of two and divides the entire sequence of numbers in that generator's period into $d = m/2l$ disjoint sequences of length l.

The problem with such a division of the entire sequence of a generator is that d is a divisor of the modulus m. As Knuth notes [2], there are subsequence correlations due to the lack of randomness in the right-hand bits of such a sequence. We describe the situation by example. Suppose we were to take the entire sequence of 2^{46} numbers in RANF, divide it into four equal parts, and give the starting value for each part to a processor. Let these seeds be s_1, s_2, s_3, and s_4. Then, the sequence starting with s_3 differs from the s_1 sequence only by the single leading bit, and the sequence starting with s_2 differs from the s_1 sequence by only the two leading bits. Our conclusion is that we must either divide such a sequence up judiciously so as to avoid divisors of m or take a modulus with few divisors. The example of Frederickson *et al.* was cleverly concocted, but rather restrictive in its allowable choices for the constants for Eq. (2). We consider instead a general method for hopping through the sequence at large equal intervals [6]. (This method, although suggested independently, is similar to the leapfrog scheme of Bowman and Robinson [5]. We explain the leapfrog scheme in Section 4.)

Consider Eq. (2). It is a linear, constant-coefficient difference equation. It is a simple matter to obtain the kth term in the series explicitly in terms of the constants. In fact, it is possible to obtain expressions for the kth term in more complicated number generators using standard methods of difference-equation analysis. For the linear congruential generator, we have

$$X_k = (a^k X_0 + (a^k - 1)c/(a-1)) \bmod m, \tag{3}$$

Indeed, as Knuth [2] points out, the subsequence consisting of every kth term of our original sequence is another linear congruential sequence having the multiplier $a^k \bmod m$ and the increment $b_k \equiv ((a^k - 1)c/(a-1)) \bmod m$. Using this simple prescription, we may "hop" through our original sequence designed for a uniprocessing machine and generate a seed for each process. Thus, for our application with r random numbers per process, we choose $k = r$ and obtain a result independent of the number of processes.

The only remaining problem is how to calculate a^k and b_k efficiently. Consider first the case with $c = b_k = 0$. One could multiply $a \times a \times a ...$, but this would require the same number of multiplies as running the random number generator. Instead, we should use one of the standard methods for accelerating the process. These methods are referred to as the *binary* method and the *factor* method, based on either writing k as a binary number or factoring k [2]. As a simple example, consider k a power of two. To calculate $a^k \bmod m$, we form

$$\begin{aligned} a^2 &= a \times a && \bmod m \\ a^4 &= a^2 \times a^2 && \bmod m \\ a^8 &= a^4 \times a^4 && \bmod m \cdots \end{aligned}$$

which requires $\log_2 k$ steps. Thus, we could find the 1024th term in our linear congruential series with only 11 multiplies (10 to find a^k). The major point is to evaluate each multiply mod m so that the appropriate number of bits is retained. This is valid, since it can be shown that the multiplication mod m is associative. Once the multiplier a^k for our subsequence is known, we generate the new seeds for the remaining processors in one multiplication mod m per processor. One word of caution: often the mod function part of a random number generator is written in assembly language so that it can be done accurately when m is of the same order as the number of bits carried by the machine. If one wishes to generate a^k using such an m, the mod function for its generation must be similarly coded.

The generalization for $c \neq 0$ is straightforward. To avoid the division in Eq. (3), write b_k in the form

$$b_k = c(1 + a + a^2 + a^3 + \cdots + a^{k-1}).$$

Then evaluate the polynomial expression by a sequence of multiplies and additions mod m as before. Consider the example above. Form

$$\begin{aligned} p_1 &= 1 \\ p_2 &= p_1 a + p_1 = 1 + a \\ p_3 &= p_2 a^2 + p_2 = 1 + a + a^2 + a^3 \\ p_4 &= p_3 a^4 + p_3 = 1 + a + \cdots + a^7. \end{aligned}$$

If the procedure for computing a^k with k any integer requires $l(k)$ multiplications mod m, then calculating b_k requires an additional $l(k)$ multiplications and $l(k)$ additions (both mod m) [2].

4. Applications

For our application, we conducted a study of the best way to group the realizations into tasks or processes, given the computational environment. One might believe that within the time-sharing-combined-with-multitasking system [7], it would be efficient to create many shorter length tasks to improve load-balancing on the system. However, testing showed that creating a significantly larger number of tasks than the available number of processors can lead to very large memory charges. Thus we determined it best to group the realizations into two tasks for the two-processor machine and four tasks for the four-processor machine. Since the total number of realizations is known in advance, a starting seed for each task can be determined (by hopping through the sequence to find the appropriate seed for the given task size) which will ensure that each multitasking machine receives the same total set of random numbers as a unitasking machine does.

The scheme for random number generation is particularly well suited to our problem, since the order in which the random numbers are assigned to a given realization does not vary with the number of processes. Since we are using a combination of uniform random numbers to determine a Gaussian random number, this guarantees reproducibility of the runs as we switch machines.

Suppose one does not know a *priori* how many random numbers will be needed per process. It is relatively simple to pick a suitable upper bound for each process. Consider the following example for the RANF generator. We wish to divide up the usual sequence while missing most of the obvious powers of two. We choose a hopping increment of 3^{20}, which falls approximately midway between 2^{30} and 2^{31}. This divides the sequence into roughly 20,000 pieces. Using a power method, we can compute the multiplier that will generate our subsequence of seeds in 2×20 multiplies mod m, i.e.,

$$a^3 = a^{3^0} \times a^{3^0} \times a^{3^0}$$

$$a^9 = a^{3^1} \times a^{3^1} \times a^{3^1} \cdots$$

Computing forty mod m multiplies is insignificant when compared with the minute of CPU time it would take to calculate each seed using a (vectorized) random number generator that calculates 64 random numbers per microsecond. This also shows that it is unlikely any application would have need of a larger number of random numbers.

In the first application we discussed, the quality of the random numbers is obviously equal to that provided by normal use of the RANF function. It was sufficient for our application. If a higher degree of randomness is desired, we suggest that the random numbers be mixed by a shuffling procedure such as the one discussed in Press *et al.* [8]; however, it may not be possible to guarantee reproducibility in this case. In the second application (i.e., using 3^{20}), the quality is likely similar, since we have avoided the obvious powers of two.

An alternate way to tackle problems that cannot specify the number of random numbers needed per process is to use the leapfrog method of Bowman and Robinson [5]. The primary difference between their scheme and ours lies in deciding which to use as a left method and which as a right. In the leapfrog scheme, the seeds (left method) are generated by a standard generator, and the strings (right method) are given by a hopping multiplier. Thus, each of the k processors possesses a multiplier that skips over k elements in the sequence to determine the next random number. (This is the same way that a vectorized random number generator works—refilling a table in groups of 64, say.) The problem with applying the leapfrog scheme to our case is that the Gaussians would change as the numbers of processors changed. We feel that our scheme may be slightly easier to implement, since the basic number generator never changes, only the seeds. However, the leapfrog method may contain a lesser number of intersequence correlations, since it would sample a smaller, contiguous set of the original random number sequence.

5. Summary

We have provided a means for generating reproducible sets of pseudo-random numbers for use in Monte Carlo codes run on multiple-processor computers. The method is based on determining the subsequence which hops through the original sequence in large steps—one step per processor. We have suggested using a fact method for evaluating powers to determine the constants of the subsequence.

Acknowledgments

We gratefully acknowledge discussions with Dr. Mark Durst and thank Dr. W. R. Martin for bringing the leapfrog method to our attention. We also acknowledge useful comments by Drs. K. Fong and G. Hulbert. This work was supported by the U.S. Dept. of Energy at Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

References

1. A. E. Koniges and C. E. Leith, *Phys. Fluids* **30**, 3065 (1987).
2. D. E. Knuth, *The Art of Computer Programming*, Vol. 2 (Addison–Wesley, Reading, MA, 1981).
3. P. Frederickson, R. Hiromoto, T. L. Jordan, B. Smith, and T. Warnock, *Parallel Comput.* **1**, 175 (1984).
4. W. R. Martin, in *Proceedings, Sixth IMACS International Symposium on Computer Methods for Partial Differential Equations, Bethlehem, PA, 1987*), edited by R. Vichnevetsky and R. S. Stepleman (Publ. IMACS, New Brunswick, NJ, 1987), p. 487.
5. K. O. Bowman and M. T. Robinson, in *Proceedings Second Conference on Hypercube Multiprocessors, Knoxville 1986*, edited by M. T. Heath (SIAM, Philadelphia, 1987), p. 445.
6. A. E. Koniges and C. E. Leith, in *Proceedings, 12th Conference on the Numerical Simulation of Plasmas, San Francisco, 1987*, p. pm 24.
7. D. V. Anderson, E. J. Horowitz, A. E. Koniges, and M. G. McCoy, *Comput. Phys. Commun.* **43** 69, (1986).
8. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes* (Cambridge Univ. Press, Cambridge, 1986), p. 194.

Received: October 5, 1987; revised: May 3, 1988

A. E. Koniges and C. E. Leith
Lawrence Livermore National Laboratory
Livermore, California 94550