from *SIAM News*, Volume 33, Number 9

Pseudo-random Number Generation for Parallel Monte Carlo—A Splitting Approach

By Roy Wikramaratna

Monte-Carlo simulations are common and inherently well suited to parallel processing, thus requiring random numbers that are also generated in parallel. We describe here a splitting approach for parallel random number generation. Various definitions of the Monte-Carlo method have been given. As one example [1]: "The Monte-Carlo method is defined as representing the solution of a problem as a parameter of a hypothetical population, and using a random sequence of numbers to construct a sample of the population from which statistical estimates of the parameters can be obtained." The method has been defined more broadly [2] as "any technique making use of random numbers to solve a problem." A more encompassing description [6] is "a numerical method based on random sampling." In this article we take the definition in its most general sense, and it is the random aspect of Monte Carlo that is our focus.

Monte-Carlo simulation consists of repeating the same basic calculation a large number of times with different input data and then performing some statistical analysis on the set of results. Input data for the different "trials'' are selected using values in prescribed distributions, using a pseudo-random number generator. The basic computation typically involves a significant amount of calculation, so that the pseudo-random number generation itself represents a small fraction of the total computational effort.

APPLICATIONS ON ADVANCED ARCHITECTURE COMPUTERS

Greg Astfalk, Editor

In theory, the accuracy of the method improves with the number of trials. In practice, this improvement is dependent on the quality of the pseudo-random number generator used. The obvious approach to parallelization—which has led to the description of Monte-Carlo calculations as "inherently parallel" or "naturally parallel"—involves simply assigning each trial to any available processor. Provided that the number of trials is large compared with the number of processors available, this approach can lead to efficient parallel computation. This is so even when the computational effort varies from one trial to another. When computational speeds also vary significantly among processors, a slightly more sophisticated approach to scheduling may be desirable. However, the basic approach remains the same.

The success of any Monte-Carlo application depends crucially on the quality of the pseudo-random number generators used. To achieve the theoretical convergence rates associated with the method, the pseudo-random number generators must have certain properties. Both the quality of the generators and the statistical independence of the results calculated on each processor are important.

A recent article in *SIAM News* [5] distinguished between two alternative approaches to the parallelization of Monte-Carlo applications. With the first approach, parameterization, a family of random number generators is defined by a recursion containing a parameter that can be varied. Each valid value of the parameter leads to a recursion that produces a unique, full-period stream of pseudo-random numbers that can be used on a particular processor. By using a different parameter value on each of P identical processors, it is possible to undertake Monte-Carlo-type calculations in parallel, with a speedup approaching P. The benefits of paral-lelization can be realized only when the results calculated for each processor are statistically independent, i.e., when the streams of random numbers generated on each processor are independent.

As the number of processors increases, additional valid parameters are needed; even if consideration is limited to pairwise independence of the random number generators, it is necessary to consider $P(P + 1)/2$ such pairs of generators for an implementation on P processors. Three different methods for parallel random number generation were considered in [5]: linear congruential generators, shift-register generators, and lagged-Fibonacci generators. All of these generators are amenable to various forms of parameterization.

The second approach, which was not considered in [5], is splitting. In splitting, the output from a single random number generator with a long period is split into a number of substreams. The substreams are then used either on different processors or for different trials in the Monte-Carlo calculation. For the splitting approach to be feasible, the underlying generator must satisfy several conditions:

1. The generator must pass appropriate theoretical and/or empirical tests of randomness. This is equivalent to the requirement that would be placed on each individual generator in a parameterized approach to parallel generation or on a single generator in serial applications.

2. The period length of the generator must be sufficiently long that all the realizations can be carried out without exhausting the available sequence. As a result, the period length required can be significantly longer for splitting than for the parameterization approach. In the latter case a separate generator is defined for each processor, which consequently supplies only a fraction of the total requirement.

3. An efficient algorithm is required for initializing the random number generator for each subsequence.

4. An efficient algorithm is required for stepping through each subsequence.

Additionally, suppose that it is possible to establish a bound on the length of the subsequence, *t*, required for each realization. The first subsequence is chosen to consist of the first *t* consecutive numbers in the sequence, the second subsequence to consist of the next *t* consecutive numbers in the sequence, and so on. If each realization is assigned to the next processor that becomes free, then it becomes feasible to undertake the identical Monte-Carlo calculation either on a serial machine or on an arbitrary number of parallel processors. This holds regardless of the relative computational capacity or availability of the different processors.

Although an absolute bound on the number of random variates required in a particular realization is not always possible, it may be possible to establish an approximate bound that will rarely be exceeded in practice. Under these circumstances, the above approach can still be used, provided that there is a well-defined way to extend the subsequence in the few cases in which the approximate bound is exceeded.

This article describes an approach, known as additive congruential random number (ACORN) generators, that is amenable to parameterization and is also particularly well suited to splitting. The ACORN generators give rise to sequences that approximate being uniformly distributed in up to *k* dimensions, for any given value of *k*. They can be selected to give sequences with period lengths in excess of any given number—specifically, in excess of 2^{30p}, for any value of *p*. Implementation is straightforward in any high-level language; if integer arithmetic is used, identical sequences can be generated on any machine. In addition to providing an extremely efficient way to calculate the next number in the sequence given the current state of the generator, they offer an efficient algorithm for stepping through the sequence in strides of arbitrary length. This latter feature allows efficient initialization of subsequences for use on different processors. Finally, there is a natural extension to a related family of generators that can be used to complete any realization for which the specified subsequence is of insufficient length. This feature is particularly useful when the number of random variates required for each realization can be bounded only in an approximate sense.

ACORN Generators

The *k*th-order ACORN generator is defined [9, 10] from an integer modulus *M*, an integer seed $Y_0^0 (0 < Y_0^0 < M)$, and an arbitrary set of *k* integer initial values Y_0^m, $m = 1, \ldots, k$, each satisfying $0 \le Y_0^m < M$, by

$$Y_n^0 = Y_{n-1}^0 \qquad n \ge 1 \tag{1}$$

$$\begin{aligned} &Y_n^m = \left(Y_n^{m-1} + Y_{n-1}^m\right) \bmod M \\ &n \ge 1, m = 1, \ldots, k \end{aligned} \tag{2}$$

$$X_n^k = Y_n^k / M \qquad n \ge 1 \tag{3}$$

If a few simple constraints on the initial parameter values are satisfied, the numbers X_n^k defined by (1)–(3) appro-ximate being uniformly distributed on the unit interval, in up to *k* dimensions. The modulus *M* needs to be a large integer, and the seed Y_0^0 and the modulus should be chosen to be relatively prime. For example, it is convenient to take $M = 2^{30p}$ for some small integer *p* and to choose Y_0^0 to be an odd integer.

Implementation

The ACORN random number generator is extremely simple to implement in any high-level language, requiring fewer than twenty executable lines of Fortran. A sample implementation, which can be used for values of the modulus up to 2^{60} (corresponding to a choice of $p = 2$), is available from the author. The implementation is based on [11]. Extension either to larger values of the modulus or to higher orders is straightforward. Analogous implementations have been made in both C and C++.

Empirical Testing

Extensive empirical testing, including that documented in [9], has demonstrated that the numbers X_n^k are uniformly distributed in the unit interval [0, 1) and satisfy a range of statistical tests of randomness. It has been shown that increasing the order also improves the randomness. Based on experience, it is recommended that a generator of at least order 10 be used. It has also been observed that increasing the modulus improves the randomness of the generator; for a practical implementation, a modulus equal to 2^{30p}, for a small integer value of *p*, appears to be a reasonable choice.

Theoretical Results

It is shown in [10, 11] that the numbers Y_n^m are of the form:

$$Y_n^m = \left(\sum_{i=0}^{m} Y_0^i Z_n^{m-i} \right) \bmod M \tag{4}$$

where

$$Z_n^{m-i} = \frac{(n+m-i-1)!}{(n-1)!(m-i)!} \tag{5}$$

The theoretical analysis in [10] shows that in fact the *k*th-order ACORN generator approximates being uniformly distributed in all dimensions up to *k*. This is in the sense that the *j*-tuples $(X_n^k,\ X_{n+1}^k, \ldots,\ X_{n+j}^k)$ approximate being uniformly distributed in *j* dimensions for each $j < k$.

For contrast, it is worth mentioning the analogous result for the linear congruential generator, which is widely used as a source of uniformly distributed random numbers and is one of the generators considered in [5]. A linear congruential generator is defined by:

$$V_n = (aV_{n-1} + d) \bmod M \tag{6}$$

$$U_n = V_n / M \tag{7}$$

where $0 < a < M$ and $0 \leq d < M$, and the U_n are the desired pseudo-random numbers defined in the unit interval. The special case $d = 0$ (known also as the multiplicative congruential generator) was proposed originally in [4]. The more general case of non-zero *d*, due independently to [7] and [8], is often called the mixed congruential generator. It can be shown [9] that a linear congruential generator approximates being uniformly distributed in one dimension, but not in any higher dimensions. In practice, for a linear con-gruential generator, it is necessary to make a very careful choice of the combination of parameters *a*, *d*, and *M* in order to ensure a reasonable approximation to uniformity in higher dimensions. The choice of good combinations of parameters is a nontrivial problem that has been very widely studied; it is dealt with at some length in [3].

Periodicity of an ACORN Sequence

It has been shown [9] that the period length of an ACORN sequence with a modulus equal to a power of two will be an integer multiple of the modulus, provided only that the seed is chosen to be odd. This means that the period length of the sequence can be increased, effectively without limit, simply by increasing the value of the modulus by a suitable factor and then choosing an odd value for the seed. As discussed elsewhere in this article, extension of the ACORN algorithm to modulus values of 2^{90}, or even 2^{120}, is straightforward.

This result contrasts with the case of the linear congruential generators, where the period length can never exceed the modulus. Furthermore, increasing the modulus for a linear congruential generator is a nontrivial exercise. This is a consequence of the time-consuming process of identifying appropriate new values of the parameters *a* and *d* (in (6)) in order to ensure reasonable distribution properties in higher dimensions. Implementation of a linear congruential generator also becomes progressively more complicated with increasing moduli.

Arbitrary-length Strides Through an ACORN Sequence

It is possible to rewrite (4) as

$$Y_{j+n}^m = \left(\sum_{i=0}^{m} Y_j^i W_n^{m-i} \right) \bmod M \tag{8}$$

where

$$W_n^{m-i} = \left(Z_n^{m-i} \right) \bmod M \tag{9}$$

By calculating the numbers W_n^{m-i} for a given value of $n = s$ (the initialization step), it becomes possible to calculate strides of an arbitrary length *s* through an ACORN sequence (the stride step) by making use of (8), provided only that it is possible to carry out both multiplication and addition modulo *M*.

For the initialization step, an obvious way to calculate W_n^{m-i} for any given *n* is to initialize an *m*th-order ACORN generator with a seed equal to one and all initial values zero and apply (1) and (2) *m* times; in this case, the W_n^{m-i} are precisely the Y_n^{m-i}. Although the method works well for small or moderate values of *n*, it becomes somewhat inefficient as *n* becomes larger, requiring a time approximately equal to *n* calls to the ACORN algorithm. For large *n* it becomes more efficient to apply an algorithm that takes advantage of the fact that

$$W_{2j}^m = \left(\sum_{j=0}^{m} W_j^i W_j^{m-i} \right) \bmod M \tag{10}$$

This is equivalent to taking a stride step of length *j* through an ACORN sequence initialized with a seed W_j^0 and initial values W_j^i, $i = 1, \ldots, m$. The stride algorithm has been implemented for a modulus of 2^{60}. Care must be taken in order to perform both the integer addition and the integer multiplication modulo 2^{60}, required in the stride step, while avoiding overflow. The extension of this stride algorithm to arbitrary moduli is straightforward in theory, although the details of the implementation become somewhat more complicated than for the ACORN algorithm itself.

Extension of Subsequences

The stride length *s* selected should ideally be at least as large as the maximum number of variates that will be required in any single realization. If it is not possible to establish an absolute bound on the number of variates required, then the stride length should

be large enough to cope with the majority of cases. In the relatively rare event that the specified stride length is insufficient, we need to extend the subsequence in a way that avoids reuse of the initial part of the next subsequence. It turns out that the most natural and straightforward approach is to simply reduce the order of the sequence by one and continue using the ACORN algorithm with reduced order. The sequence actually used in this case would then consist of the numbers X_m^k, X_{m+1}^k, . . . , X_{m+s-1}^k, X_{m+s}^{k-1}, X_{m+s+1}^{k-1}, ... In the exceptional event that more than $2s$ variates were required for a single realization, the order could once again be reduced and the sequence continued as before. This approach can be adopted in either a serial or a parallel implementation.

Computational Performance

The computational performance of the ACORN algorithm is comparable with that of a linear congruential generator that has a similar period length. Owing to the simplicity of the algorithm, it is feasible to improve its computational speed. One approach is in-line coding; another is to modify the implementation by tuning it for specific hardware.

It is considered unlikely that the generation of the random numbers will consume more than a small portion of the total execution time in a Monte-Carlo calculation. Thus, the benefits of extensive code modifications to ACORN are likely to be marginal. The time taken to generate a single random variate has been seen to be proportional to the order k of the ACORN generator. For an implementation with a modulus $M = 2^{30p}$, the time taken to generate each random variate is proportional to p.

The stride algorithm can be coded (for $M = 2^{60}$) so that the time taken for a stride of arbitrary length is of the order of several hundred calls to the ACORN generator. When the stride length exceeds a few thousand random variates, considerable savings can be achieved by parallelization of the code.

A similar approach can be adopted when the stride length is shorter, but in such cases it may be more efficient to calculate a number of realizations, say 10 or 100, at a time on each processor, with a stride length that has been increased by a corresponding amount. For practical problems the time spent in generating the random variates generally represents less than, say, 10% of the total computational effort. When the total number of realizations is large compared with the number of processors P, the speedup obtained can approach the absolute theoretical maximum of P.

By adopting this approach, in which a separate pseudo-random subsequence is defined for each realization, it is possible to perform the computations efficiently in parallel, even if the speed or availability of processors varies widely. The only requirement is that it should be possible to schedule each realization to run on the next processor that becomes available.

Advantages of the ACORN Approach

The method described here for generating uniformly distributed pseudo-random numbers is convenient to use as an off-the-shelf generator and offers some additional benefits when compared with the more commonly used methods.

The major advantages of the ACORN approach to random number generation are:

1. It is extremely simple to code for arbitrary modulus and order, and can be implemented in virtually any high-level programming language. It can be coded as a subroutine or function, or, for maximum computational efficiency, it can be coded in-line. Implementation as a user-callable function gives users an extremely straightforward way to access the algorithm from their own software.

2. There are very simple rules for selecting a good generator; all generators selected by these rules appear to give comparable results for given moduli and order.

3. By coding the generator in integer arith-metic, it is possible to obtain identical sequences on any machine and with any program-ming language (provided that the language will permit integer arithmetic modulo 2^{30}).

4. If the period length is found to be insufficient, it is straightforward to increase the period simply by increasing the modulus. The period length increases in proportion to the modulus, while for large moduli the execution time per random number generated increases only in proportion to the logarithm (base 2) of the modulus.

5. Increasing the order of the generator increases the randomness of the sequence (in the sense that a kth-order ACORN generator approximates being uniformly distributed in up to k dimensions). This leads to a simple way to test the adequacy of the sequence for the particular application: repetition of the whole Monte-Carlo simulation with a generator of higher order. If the results of the simulation are the same, in a statistical sense, this suggests that the original generator was good enough. Significantly different results are strong evidence that the original generator was insufficiently random or that the number of realizations was insufficient. In the latter cases, the calculation should be repeated with generators of successively higher order and increasing numbers of realizations until the results of the calculation no longer change significantly.

6. It is possible to take strides of arbitrary length through the sequence; the time is equivalent to that required for calculating a few hundred random numbers for an arbitrary stride length. This provides a way to initialize many generators for use in a parallel application at relatively small computational cost. The benefits of this approach become increasingly significant when the stride length increases, because the time required to compute a single stride is independent of the stride length. The algorithm has been implemented in practice for a generator with modulus 2^{60} in both Fortran and C++. The algorithm can be extended to generators with larger moduli of the form 2^{30p} for arbitrary integers p, and it could be equally well implemented in other programming languages.

Conclusions

This article has outlined a possible approach to the parallelization of Monte-Carlo calculations, based on a splitting approach to the generation of parallel streams of pseudo-random numbers. Some potential advantages of splitting over the alternative of parameterization have been identified, most notably the fact that the identical set of calculations is carried out irrespective of the number of processors used. It is therefore self-evident that if the method converges in a serial implementation, it will also converge in a parallel implementation.

The ACORN generator has been proposed as a suitable pseudo-random number generator for use in conjunction with the splitting approach. It appears to satisfy all the requirements for a general-purpose uniform random number generator in serial applications. The potentially unlimited period length, coupled with the ability to calculate strides of any desired length through an ACORN sequence, also makes it particularly attractive for use in parallel applications, leading to the prospect of parallel Monte-Carlo-type calculations that are truly scalable to arbitrary numbers of processors.

The generator has proved amenable to theoretical analysis, leading to results concerning the periodicity and the uniformity of the resulting sequences. Further theoretical analysis of the algorithm could lead to further improvements in our understanding of this approach to pseudo-random number generation.

The algorithms have been successfully coded and tested. Experience to date with the ACORN generator has been very encouraging. The simplicity of the implementation makes its incorporation into existing applications that require a source of uniform random numbers straightforward. Current work is aimed toward the identification of additional real problems that will be suitable for further validating the ACORN generators and for demonstrating the speedup that can be obtained in practice on a suitable parallel machine.

References

[1] J.H. Halton, *A retrospective and prospective survey of the Monte-Carlo method*, SIAM Rev., 12 (1970), 1–63.

[2] F. James, *Monte Carlo Theory and Practice*, Rep. Prog. Phys., 43 (1980),1145–1189.

[3] D. Knuth, *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms, Addison-Wesley, Reading, Massachusetts*, 1981.

[4] D.H. Lehmer, *Mathematical methods in large-scale computing units*, Proceedings of the 2nd Symposium on Large-scale Digital Calculating Machinery, Cambridge, Massachusetts, 1949, Harvard University Press, Cambridge, Massachusetts, 1951, 141–146.

[5] M. Mascagni, *Parallel pseudorandom number generation*, SIAM News, 32 (5) June 1999.

[6] H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*, SIAM, Philadelphia, 1992.

[7] A. Rotenburg, *A new pseudo-random number generator*, J. Assoc. Comput. Mach., 7 (1960), 75–77.

[8] W.E. Thompson, *A modified congruence method of generating pseudo-random numbers*, Comput J., 1 (1958), 83–86.

[9] R.S. Wikramaratna, *ACORN—A new method for generating sequences of uniformly distributed pseudo-random numbers*, J. Comput. Phys., 83 (1989), 16–31.

[10] R.S. Wikramaratna, *Theoretical background for the ACORN random number generator*, AEA Petroleum Services Report AEA–APS–0244, 1992.

[11] R.S. Wikramaratna, *Theoretical analysis of the ACORN random number generator*, SIAM Conference on Applied Probability in Science and Engineering, New Orleans, Louisiana (unpublished AEA Petroleum Services internal report PRTD (90)R62), 1990.

Roy Wikramaratna (roy.wikramaratna@aeat.co.uk) is a technical manager with AEA Technology plc, at Winfrith in Dorset, UK. His research interests are in mathematical, numerical and computational methods with applications in oil reservoir engineering.