JOURNAL OF COMPUTATIONAL PHYSICS 64, 271–275 (1986)

Note

A Method for Vectorized Random Number Generators*

In this brief note we describe a technique that we have used to generate pseudorandom numbers on a CYBER–205 at an average rate of one per minor cycle. Our aim is to generate the identical sequence of random numbers which is produced by the CYBER–205 intrinsic subroutine VRANF, but at a faster rate. The general principles can be implemented on any SIMD machine, including array processors and CRAY's. We do not here consider the issue of statistical properties of random number generators (rng's) but rather will discuss several features having to do with vectorization of the linear congruential algorithm,

$$X_n = (aX_{n-1}) \qquad \text{mod } M, \tag{1}$$

The modulus, M, is often chosen in order to take advantage of machine architecture. So, for example, the CYBER–205 Fortran rng uses $M = 2^{47}$, corresponding to a 47-bit mantissa.

The iterative nature of Eq (1) appears at first to prohibit vectorization (i.e., a parallel algorithm). Upon further scrutiny it turns out that vectorization is possible provided that one first creates a vector of seeds, "X_n," *or* a vector of multipliers, "a." The first technique—that of using a seed-buffer—is presently in use in the CRAY Fortran libraries, and has been used on the CYBER–205 [1, 2]. We will discuss here the alternate method of multiplier-vectors.

The scalar algorithm consists of the following steps:

(S1) Pick a fixed-point default seed, MSEED, and a fixed-point multiplier, MULT.

(S2) Transform MSEED to a floating-point seed, RSEED, by packing "-47" into the exponent.

(S3) Multiply RSEED by the multiplier and take the lower half of the product. On the CYBER–205, this operation picks out a mantissa consisting of the lower 47 bits of the mantissa-product, and an exponent of -47, the product of the two exponents. The result is the new floating-point seed, RSEED.

(S4) Normalize the result by adding 0.0. This value is the output of the random number generator.

* Work supported in part by the NSF.

0021-9991/86 $3.00

Our vector algorithm is similar:

(V1) The starting seed is created as in steps S1 and S2 above.

(V2) Then generate a multiplier-buffer of length L, where the nth element is defined by

$$a(n) = [a(n-1) \cdot a] \mod M \tag{2}$$

and "a" is the multiplier, MULT. On the CYBER–205 the modulus is implemented, as described above, by taking the lower half of the product. The value of L should be chosen large enough so that the vector start-up time is negligible. In our program we use $L = 65535$, which is the maximum allowable vector length. The multiplier-buffer never needs to be updated.

(V3) Next, construct a random number vector, of length N, by linking the lower-product and normalization steps. Namely, one first creates a vector of seeds by multiplying RSEED with the elements of the multiplier-buffer, extracting the lower half. This instruction is "linked" to the following step, which is the normalization step S4 above.

(V4) The procedure is terminated by fetching the new seed to be used for the next set of random numbers. This is a single scalar operation consisting of lower-multiplication of RSEED by $a(N)$.

The resulting algorithm produces precisely the same sequence of pseudorandom numbers as the system-supplied subroutine VRANF, which is an optimized scalar routine. (However, it should be noted that in any two subsequent calls to VRANF, one random number is skipped. That is, if each call produces a vector of length N, the second vector begins at the $(N+2)$nd number in the sequence.) Some important features should be mentioned. First of all, the length N of the desired random number vector can be smaller than the size of the multiplier-buffer. If N is less than L, then only the first N multipliers are used. Furthermore, it is not necessary to use the same value of N each time the random numbers are needed. This feature makes it easy, in principle, to interweave scalar and vector random number generation. Second, only two arithmetic operations are required in vector mode (step V3). On the CYBER–205, each of these operations consists of a single machine instruction, and both instructions can be linked so that they execute in parallel. This results in a long-vector speed of one random number per cycle. Both of these features distinguish our algorithm from the seed-buffer algorithm that has been used previously. We believe that we can gain a factor of two in speed over that algorithm. There also may be several advantages to having a fixed buffer of multipliers with flexible-length random number vectors.

It is worth observing here that the CYBER linear congruential algorithm has been criticized by Kalle and Wansleben [3]. They note that in several large Monte

Carlo applications, results appear to be dependent on the generator. They recommend instead a certain shift-register generator. The advantage of that method may be less obvious for simulations on systems whose sizes are not large powers of 2, or whose interactions are more complex than the Ising Model interactions (which do not require many floating-point operations) considered in Ref. [3], or for other update methods, such as the heat-bath update used in $SU(2)$ gauge theories. Indeed, the linear congruential method may be more efficient than the shift-register method for problems which require short vectors of pseudorandom numbers, or which interleave scalar and vector generation. One way in which we might be able to improve upon the statistical properties of the CYBER linear congruential algorithm, at no cost in time, is by using an additive term in the generator, namely,

$$X_n = (aX_{n-1}) \bmod M + b \bmod M. \tag{3}$$

That can be easily implemented by changing step S4 above (and its vector counterpart) so that $b \pmod M$ is added instead of 0.0 (for the vector version we would have to initialize an adder vector). We have not further explored this technique. Nor have we considered the statistical properties of the half-precision analogue of Eq.(1). The CYBER–205 supports half-precision operations which would allow the linear congruential method to be used with a modulus of 2^{23}. This algorithm could go at approximately double the speed of the one we have studied. Work would be required to determine good choices of multiplier, starting seed, and possibly additive term. In the meantime, for applications which require half-precision pseudorandom numbers, we proceed as follows: first generate full precision numbers using RAND (V) (the algorithm described in steps V1–V4) and then using the special Fortran call Q8CONV convert the resulting vector to half-precision. (Alternatively, in Fortran, one can simply assign a half-precision vector to the result of RAND (V). This implicity performs the conversion Q8CONV.)

We conclude by presenting a tested CYBER–205 random number subroutine. Much of it consists of special Fortran calls ("Q8...") which directly translate into machine instructions. This routine has been designed with the idea that each program will make many calls to RAND, each of which will produce very long vectors. Therefore little attention has been paid to reducing the routine-calling overhead, or to optimizing the speed of the initialization segment of the routine. So, for instance, one call to RAND, with an argument vector of length 3000, takes 43 μsec on a 2-pipe CYBER–205. The overhead is approximately 10 μsec. Thus the speed is about 10 μsec per number with an overhead of about 1000 numbers. For comparison, the CYBER–205 intrinsic routine VRANF takes about 350 nsec per number. Of course, the first call to RAND also results in the initialization of the multiplier buffer. This could be made more efficient by first initializing a small buffer, and then using that in a recursive vector operation to generate the rest of the buffer. Also, initialization time can be decreased if the buffer length is reduced from its maximum of 65535. In fact, that would be desirable if the buffer vector is subject to paging (which costs more than 1 μsec per multiplier).

```
      SUBROUTINE RAND(V)
C
C THIS ROUTINE GENERATES A RANDOM VECTOR, STARTING WITH THE MOST RECENT
C SEED.  THE LENGTH OF THE VECTOR IS THE LENGTH OF THE DESCRIPTOR, V.
C ENTRY POINTS 'RANDSET' AND 'RANDGET' ARE SIMILAR TO THE CYBER-205
C 'RANSET' AND 'RANGET'.  THE MULTIPLIER AND DEFAULT SEED ARE THOSE OF
C THE CYBER-205. THE FIRST CALL TO THIS SUBROUTINE INITIALIZES THE
C MULTIPLIER VECTOR.
C
      COMMON/R8RAN/R8SEED,MR8SEED,R8ZERO,MR8MULP,MR8,MR8EXP,MR8ZERO
      COMMON/R8MULV/MR8MULV(65535)
      DESCRIPTOR V, MR8MULVD
      DATA R8ZERO/0.0/,MR8MULP/X'00004C65DA2C866D'/,MR8/0/,MR8EXP/-47/,
     1MR8SEED/X'000054F4A3B933BD'/,MR8ZERO/0/
C
C
      IF (MR8.EQ.1) GO TO 200
      CALL Q8PACK(MR8EXP,MR8SEED,R8SEED)
      MR8MULV(1)=MR8MULP
      DO 10 I=2,65535
 10   CALL Q8MPYL(MR8MULV(I-1),MR8MULP,MR8MULV(I))
      MR8=1
 200  CONTINUE
      GO TO 100
      ENTRY RANDSET(ISEED)
      IF (MR8.EQ.1) GO TO 300
      MR8MULV(1)=MR8MULP
      DO 310 I=2,65535
 310  CALL Q8MPYL(MR8MULV(I-1),MR8MULP,MR8MULV(I))
      MR8=1
 300  CALL Q8PACK(MR8EXP,ISEED,R8SEED)
      RETURN
 100  CONTINUE
      CALL Q8LTOR(V,,L)
C
C SET UP REGISTERS FOR LINKED TRIAD
      ASSIGN MR8MULVD, MR8MULV(1;L)
      CALL Q8RTOR (R8SEED,,3)
      CALL Q8RTOR (R8ZERO,,4)
      CALL Q8RTOR (V,,5)
C
C EXECUTE LINK INSTRUCTION
      CALL Q8LINKV(X'8')
      CALL Q8MPYLV(X'10',,3,,MR8MULVD,,5)
      CALL Q8ADDNV(X'10',,4,,5,,5)
C
      CALL Q8MPYL(R8SEED,MR8MULV(L),R8SEED)
      RETURN
      ENTRY RANDGET(NEWSEED)
      CALL Q8PACK(MR8ZERO,R8SEED,NEWSEED)
      RETURN
      END
```

ACKNOWLEDGMENTS

We gratefully acknowledge the help we were given by R. Begin, especially in setting up the link-triad. We would also like to thank D. Barkai for explaining the seed-buffer method, our referees for a number of interesting and useful suggestions, and L. M. Thorndyke, B. Robertson, and L. K. Steiner of ETA systems, Inc., for access to the ETA Systems 4 Mword 2 vector pipeline CDC CYBER-205.

REFERENCES

1. D. BARKAI, K. J. M. MORIARTY, AND C. REBBI, *Comput. Phys. Commun.* **32** (1984).
2. A. D. KENNEDY, J. KUTI, AND B. J. PENDLETON, NSF–ITP–84–62, 1984.
3. C. KALLE AND S. WANSLEBEN, *Comput. Phys. Commun.* **33**, 343 (1984).

RECEIVED January 15, 1985; REVISED August 14, 1985

WILLIAM CELMASTER

Physics Department
Northeastern University
Boston, Massachusetts 02115

K. J. M. MORIARTY

Institute for Computational Studies
Department of Mathematics, Statistics, and Computing Science
Dalhousie University, Halifax, Nova Scotia B3H 4H8, Canada