Techniques for Testing the Quality of Parallel Pseudorandom Number Generators*

Steven A. Cuccaro Michael Mascagni Daniel V. Pryor

Abstract

Ensuring that pseudorandom number generators have good randomness properties is more complicated in a multiprocessor implementation than in the uniprocessor case. We discuss simple extensions of uniprocessor testing for SIMD parallel streams, and develop in detail a repeatability test for the SPMD paradigm. Examples of the application of these tests to an additive lagged-Fibonacci generator are also given.

1 Introduction

Random numbers are used in applications ranging from scientific simulations to statistical sampling procedures. The impracticality in most cases of using true random numbers led to the development of pseudorandom number generators (PRNGs). In the course of this development, testing procedures were designed to ensure that the necessarily deterministic sequence of numbers produced by these PRNGs had analytical and statistical properties which compared well with those of a true random stream. With the tests described by Knuth [1] and Marsaglia [2, 3], an intelligent user can be reasonably confident that he can pick a generator suitable for his purposes.

However, the tests described in the above references presuppose that only a single stream of "random" numbers is needed. To run efficiently, a program designed for multiprocessor computers will use multiple PRNGs to generate many streams simultaneously. Indeed, several methods for generating these multiple streams have been proposed [4, 5, 6, 7]. This parallelization produces new difficulties in ensuring random behavior. While multiple true random streams would not have mutual correlation under any circumstances, this is not guaranteed for PRNGs on a multiprocessor system.

In this paper, we describe some new requirements on PRNGs which arise from parallelization. Simple rules that a generator must obey to satisfy these requirements are given where possible; for the cases where there are no simple rules, or if the details of the generator are not known to the user, we outline algorithms which can be used to test that the generator behaves in the necessary manner. We also give some examples of these tests as applied to the parallel pseudorandom number generator developed by the authors [4], which is based on additive lagged-Fibonacci generators.

2 Single Processor Properties

Before we can discuss testing of parallel PRNGs, we must first determine the properties which ensure that the generators have good behavior. We start by defining the desired

*Supercomputing Research Center/I.D.A., Bowie, MD, USA

properties of PRNGs in the most constrained case of parallel processing, which is the uniprocessor case (equivalent to a collection of processors all simultaneously performing the same operations on the same data). We can then relax the constraints successively to yield the SIMD, SPMD and MIMD paradigms, and at each level define the characteristics to be tested.

There are three basic properties that any PRNG should satisfy. The first of these is of course "randomness": the stream of numbers produced should be similar to a truly random stream. Algorithms which compare various characteristics of pseudorandom streams with the known random property are covered in detail in [1] and [2] for the single processor case, and the results of applying these tests to many standard PRNGs are given. In particular, Marsaglia subjects the additive lagged-Fibonacci generator to a suite of eight tests and finds that it passes all but the birthday spacings test (and for sufficiently long lags passes this test as well).

The second property is repeatability: precisely the same sequence of pseudorandom numbers is returned when the initializing inputs to the generator are the same. (This property is needed for testing and development of programs.) It would be difficult to unknowingly design a generator without this property on a single processor, and testing is trivial.

The third property is portability: given the same initial conditions the stream should be the same on a variety of platforms. Portability is easily tested by running the generator on different machines and comparing the results. Alternatively, portability is ensured if the generator is either based on integer arithmetic and designed to give correct answers with 32-bit integers[1] or on integer or floating point addition/subtraction done to some fixed precision less than or equal to the precision of all machines to be used[2].

3 Requirements and Tests for SIMD

To satisfy the repeatability and portability requirements in the SIMD paradigm, we have to guarantee that the results of a set of processes which call the parallel random number generator do not change when the number of physical processors used by the calculation changes. We can ensure portability if the parallel generator depends only on user-provided parameters, by fixing all but one parameter to be the same on every process and using a process index for the remaining parameter. If the user does not have this degree of control over the generator, the best way to test for portability is simply to port the generator. Using a program which generates a fixed number of processes, each of which calls the PRNG, the condition is satisfied if the results of running this program depend only on the initial conditions of the generator, and not the architecture or the number of processors in the architecture.

The presence of multiple streams of random numbers requires us to generalize the definition of randomness. Along with the requirement that the individual streams of numbers seem random, we add the condition that the streams of random numbers generated by each process should behave as if they were independent. A necessary condition for this is that the correlation between the streams assigned to each process be small. With a small number of PRNGs, each stream may be tested against the others individually, but this is not practical when there are large numbers of processes. One way to measure the correlation

[1] Here we are assuming all architectures to use 32-bit or larger integers.

[2] It is worth noting that the differences between the streams produced on architectures in a case where this condition is not fulfilled are likely to be small, at first.

in this case is to do the standard randomness tests, but instead of examining the numbers generated by the PRNG associated with a single process at successive times, we examine the numbers generated by all processes at a single time. The numbers can be examined in any of the possible permutations of processor order, provided that the permutation chosen does not depend on the values of the numbers being tested (otherwise any generator could be made to fail any test by choosing a permutation which orders the numbers). The test can be repeated at successive times. If the number of processes is small, the entire set of numbers may be treated as one stream, otherwise the numbers generated at each time may be tested individually and the results of the tests checked for deviations from the expected value using a χ^2 or Kolmogorov–Smirnov type test [1].

This type of test will inevitably involve some communication between processors, so a choice which minimizes this is optimal. A simple example is to perform parallel runs testing in processor order, as in the FORTRAN 90 program included in Appendix A. The runs test compares the number and length of each monotonic increasing (or decreasing) sequence – a run – in the stream examined with the theoretical distribution of runs in a random stream (a run of length k occurs with probability $\frac{1}{k!} - \frac{1}{(k+1)!}$ [1]). The program in Appendix A compares the number generated in each processor to its neighbors to determine where runs end, modifies the result of these comparisons to exclude dependence due to adjacent runs, determines the size of the runs and tallies the number of runs of each length. Similar programs have been created for other tests, such as the equidistribution test and the serial test in n dimensions [1].

We have found that correct seeding of the generators is crucial to proper performance. The parallel random number generator package assembled by the authors [4] was found to fail a parallel runs test when seeded from an LCG sequence and from a scrambled version of this sequence, but passes the test when seeded from a Tausworth generator. (The currently available version of the generator is seeded correctly.) The correctly seeded generator also passes the equidistribution test and the serial test in up to 7 dimensions. It is interesting to note that the generator passes the birthday spacings test [2, 3] when examined across processors, even though the stream of numbers produced on any single processor fails this test.

4 Additional Requirements and Tests for SPMD/MIMD

There are important qualitative differences when we move on to the SPMD model. First, the various random number streams may get out of synch, which makes the model for testing correlation between streams that is given above less valid. A more accurate test is to check the correlation between streams at all possible offsets. Since this is impractical for generators with long periods, we can test at a selected number of offsets instead. A remedy for this inability to check all possible offsets is to prove theorems about theoretical measures of randomness for the method of generation used. These theoretical results often allow inferences about correlations among streams at all offsets [9]. Second, we must make sure that the structure of the generator does not hinder the repeatability of calculations. This requires that the method of assigning generators to newly spawned processes be deterministic and depend only on local information. In the SIMD case, this is not an issue, since every process executes the same instruction simultaneously, and if additional processes are created, they can be indexed in a deterministic manner and generators assigned to them as above. In a SPMD model, the order in which each process executes an instruction is not fixed, so indexing that depends on order of creation will result in spawned processes

receiving a different generator on each run of a program. (We note that the MIMD model will be no more general than this, so it will not be necessary to treat it seperately.)

To test the repeatability properties of a generator in the context of process creation, we have constructed a Monte Carlo simulation whose *only* function is to split at random, generating a pseudorandom tree while running. If the parallel generator is constructed properly, the generated tree will be independent of the number of physical processors used by the program, and of the order in which created processes are executed on these processors. A generator fails if the trees produced in two different runs with the same parameters differ. In practice, this comparison is much too laborious for any but the smallest trees. Instead, we examine the number of branch points at each level of the tree and the total lengths of the branches to make sure that they are the same for many trials. We also look at the number of processes executed on each processor at each level of the tree to ensure that they are different, for otherwise the test is superficial.

This tree generator uses an idea similar to one described in [8]. For their pseudorandom number generator, an LCG, they define two successor rules, so that each generated number has a right successor and a left successor. They assume that the right successor sequence is the one that is used most often by a process, and that the left sequence is used for splitting processes or spawning new threads. Thus the use of a standard random number generator in a single-processor sequential program corresponds to using the stream of right hand successors of a given seed. A seed for a new stream is obtained by taking a left hand successor at a particular node of the random tree.

For our purposes, if the generator to be tested has a provision for splitting (*i. e.*, a new generator is initialized in some automatic way when a new process is created), this is made use of; otherwise it is necessary to define some way of obtaining the new stream. If this definition is inappropriate the parallel generator will fail the test. Once an initial seed for the random tree is specified, the tree to be generated is given by three parameters: (i) a granularity constant, G, which determines the probability that a thread will "die"; (ii) an initial "fertility" rate, F_0, which is the level-0 probability at each time step of spawning children divided by the probability of dying; and (iii) a multiplier α for a geometric sequence describing the decline in the fertility rate as a function of the tree level. While the language used here ("fertility", "spawning children", "dying" *etc.*), suggests an attempt to model some real-world biological system, we have no such intent. We merely wish to generate random trees of various shapes and sizes with some criterion for stopping, and the language seems suited for describing that process.

At each level of the tree, the fertility rate will be

$$F_L = \alpha F_{L-1} = \alpha^L F_0 \ ; \ \text{where} \ \ 0 \leq \alpha < 1.$$

The granularity G is also the expected lifetime of each right sequence, or the expected number of steps it will take; the probability that a thread will die is a constant G^{-1} for all levels L of the tree. At each level, the right sequences of that generation (L) will spawn an expected F_L number of children. F_0 and α combine to control the spread and depth of the tree. In particular, without the restriction on α, the simulation would not end.

We have used this code to test the behavior of our lagged-Fibonacci based parallel generator [4]. For this generator, the "left successor" is provided by initializing a new generator which obeys the same equation as its parent but by virtue of its different initial conditions is on a different cycle. (The parallel generator has been designed to guarantee that all generators produced in a run are on different cycles [9].) "Right successors" are generated by stepping the generator in the usual fashion. Figures 1, which show the shapes

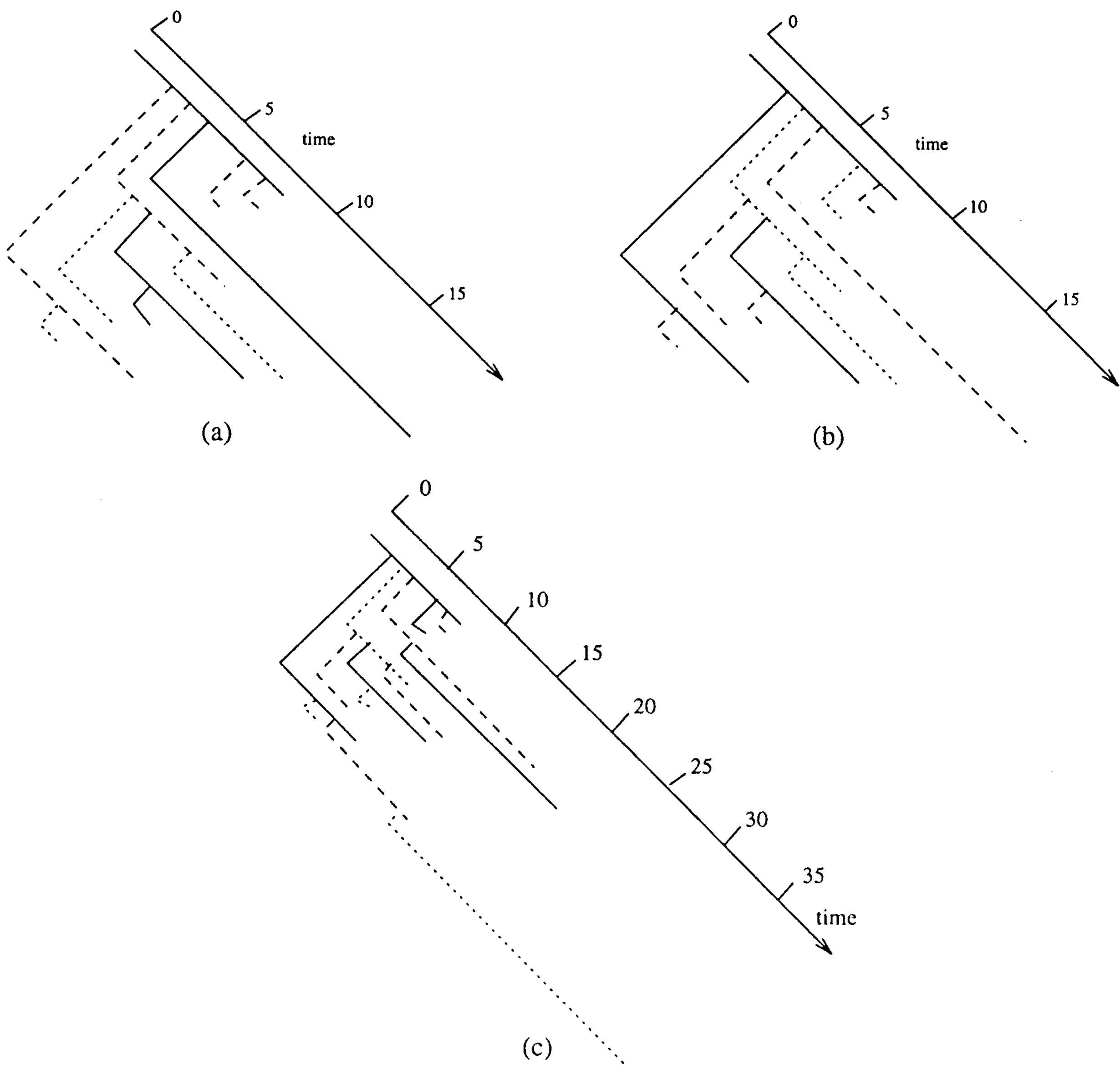

FIG. 1. *Three trees generated by the program of section x. Different styles of line represent execution on different processors. Figures 1a) and 1b) were generated with the same parameters but on two different runs; figure 1c) was generated with different parameters.*

of the partial trees generated with some sample parameters, illustrate the results of varying the parameters of the program.

5 Conclusion

We have analyzed the additional properties that are required of pseudorandom number generators for them to have good behavior when parallelized, and derived means of testing for these properties. We found that it was not sufficient merely to use a good uniprocessor generator, but that the initialization of the generators was important as well.

A FORTRAN 90 code for runs testing

```
c parameters: RUNSIZE,NPROCS
c serial integers: j, maxlen, temp, count(RUNSIZE)
c parallel integers/reals: hold, hold1
c parallel integers: context, index, ind1, ind2, kk, test
c parallel functions: pgen(), AND(), ANY(), SUM()

      do j=1,RUNSIZE
          count(j) = 0
      end do
c
c     GET RANDOM NUMBERS, COPY TO RIGHT
c
      hold = pgen()
      hold1(2:NPROCS) = hold(1:NPROCS-1)
c
      index = 0
c     RUNS UP - hold.lt.hold1
c     RUNS DN - hold.gt.hold1
      where (hold.lt.hold1) index = 1
      index(1) = 0
c
c     MAKE RUNS NON-OVERLAPPING
c
      ind1(2:NPROCS) = index(1:NPROCS-1)
      test = AND(index,ind1)
      do while (ANY(test).eq.1)
          ind2(3:NPROCS) = index(1:NPROCS-2)
          where (ind2.eq.0.and.test.eq.1) index = 0
          ind1(2:NPROCS) = index(1:NPROCS-1)
          test = AND(index,ind1)
      end do
c
c     COUNT RUN LENGTHS IN PARALLEL
c
      ind2 = index
      context = 1
      kk = 0
      maxlen = 0
      do while (ANY(context).eq.1)
            maxlen = maxlen + 1
            context(maxlen) = 0
            ind1(2:NPROCS) = ind2(1:NPROCS-1)
            where (ind1.eq.1) context = 0
            where (context.eq.1) kk = kk + 1
            ind2 = ind1
      end do
c
c     TALLY NUMBER OF RUNS OF EACH LENGTH
c
      where (kk.gt.RUNSIZE) kk = RUNSIZE
      do j=1,RUNSIZE
            temp = SUM(1,kk.eq.j.and.index.eq.1)
            count(j) = count(j) + temp
      end do
```

References

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 2nd ed. Addison-Wesley, Reading, Massachusetts (1981).

[2] G. Marsaglia, *A current view of random number generators*, in Computing Science and Statistics: Proceedings of the XVIth Symposium on the Interface, 1985, pp. 3–10.

[3] G. Marsaglia, A. Zaman, and W. W. Tsang, *Toward a Universal Random Number Generator*, Stat. and Prob. Lett. 8 (1990) pp. 35–39.

[4] D. V. Pryor, S. A. Cuccaro, M. Mascagni, and M. L. Robinson, *Implementation of a Portable and Reproducible Parallel Pseudorandom Number Generator*, to appear in proceedings of Supercomputing '94.

[5] F. W. Burton and R. L. Page, *Distributed random number generation*, J. Functional Programming 2 (1992) pp. 203–212.

[6] I. Deák, *Uniform random number generators for parallel computers*, Parallel Computing 15 (1990) pp. 155–164.

[7] R. P. Brent, *Uniform Random Number Generators for Supercomputers*, Proceedings Fifth Australian Supercomputer Conference, 1992, pp. 95–104.

[8] P. Frederickson, R. Hiromoto, T. L. Jordan, B. Smith, and T. Warnock, *Pseudo-random trees in Monte Carlo*, Parallel Computing 1 (1984) pp. 175–180.

[9] M. Mascagni, S. A. Cuccaro, D. V. Pryor, and M. L. Robinson, *A Fast, High Quality, and Reproducible Parallel Lagged-Fibonacci Pseudorandom Number Generator*, SRC Technical Report 94–115.