Implementation of a Random Number Generator for sequential and parallel computers.

Olukayode O. Odeyemi * Paul Coddington †

PROPOSAL FOR A 1995 NPAC REU PROJECT‡

Abstract

Here we propose an implementation of 64 bit sequential as well as 32 & 64 bit parallel random number generators. In the near future, 64 bit processors will be commonplace. Also parallel architectures will dominate the computing scene. This will lead to an increase in computing power, which will enable simulations of larger proportions, in particular those using random number generators. Longer periods, better randomness properties, and portability to different machines and architectures are needed for these generators to keep up with simulation requirements. The combination of two generators, efficiently implemented in a language such as High Performance Fortran, can meet these conditions.

1 Introduction

Many computer simulations e.g. Monte Carlo simulations, use random number generators. However, according to Donald Knuth [1], "Random number generators should not be chosen at random", since there are not many good generators available, which of course leads to the continuous study and improvement of random number generators. But one has to point out right away that these generators produce *pseudorandom* numbers, because the numbers generated are actually deterministic sequences, so cannot be truly random. However this is not as bad as it seems, since the numbers generated need to be deterministic and repeatable in order to debug a program that uses a random number generator.

In order for a random number generator to be acceptable, it has to have the following properties [3]:

1. Produce uncorrelated sequences.
2. Have a long period (i.e. repetitions of the sequence are far apart).
3. Sequences should be of uniform distribution.
4. The program should be efficient.

However, it is a little bit tricky to satisfy all these conditions. For example, not everything that seems random is uniform and not everthing that is uniform is random.

*Research Apprentice, 1995 NPAC REU Program, *oodeyemi@npac.syr.edu*; Electrical Engineering and Computer Science major, City University of New York (City College), *csoko@csfaculty.engr.ccny.cuny.edu*.

†NPAC Research Scientist, *paulc@npac.syr.edu*.

‡The 1995 Research Experiences for Undergraduates (REU) Program in High-Performance Computing is conducted by the Northeast Parallel Architectures Center (NPAC) at Syracuse University. The National Science Foundation provides primary funding for the NPAC REU program, through NSF Grant CDA-9424332. Additional funding is provided by Syracuse University and NPAC.

A deterministic sequence would have a hard time passing as being uncorrelated, and the other pair of conditions are difficult to achieve. Overcoming these problems is one of the tasks of this project.

One of the generators commonly used is the *Linear Congruential Generator* (LCG) [1, 2, 3, 4, 9]. These generators produce their sequence of numbers by using equations of the form:

$$S_i = A\, S_{i-1} + C \; mod \; m, \tag{1}$$

where S_i is the i^{th} seed or random number generated, S_0 being the initial seed (to be selected by user); A is some multipier (to be determined); m is a modulus (to be determined) and C is some constant. For the terms that need to be determined, there are rules and tests that help choosing them [1]. One of these tests is the *spectral test* [1, 3, 9], a popular test used to determine the randomness of the output of a generator. We would like a generator to have a uniform spread as in Figure 1, rather than having a lattice structure as in Figure 2. The spectral test properties for LCG's improve as the word size of the numbers used is increased, for example from 32 to 64 bits [1, 4, 9].

FIG. 1. *This graph shows good randomness.*

FIG. 2. *This one, however shows a pattern thus some kind of correlation.*

A combination of two LCG generators is advantageous since it provides longer cycles and better randomness properties [7, 4, 9]. This combination can be done by simply adding the numbers produced by the two LCGs, modulo the smallest value of m for the two generators. It is this kind of generator that will be used in this project. The maximum period of a linear congruential generator is m [1], which is about 10^9 for generators using 32-bit numbers and 10^{19} for 64-bit generators. Combining two LCG's gives a generator with a period which is the product of the periods of the two LCG's, i.e. 10^{19} for 32-bit generators and 10^{38} for 64-bit generators.

As time goes by, there is going to be the need for generators that are portable to different machines and architectures (particularly parallel architecture machines), for example, in order to cross-check simulation results. This would mean that most generators used right now would be obsolete in due time, thus the need for better implementations. Within the next five to ten years, large simulations on teraflop supercomputer would easily exhaust 32-bit generators we have today, thus the need for 64-bit implementations. There are some implementations of the 32-bit sequential and parallel random number generators [6], but these could be improved upon as well as extended to a 64-bit implementation, that is, an extension to the work done by Cote and L'Ecuyer[6] for 32-bit combined LCGs.

2 Proposed Work

For this project, the work to be done is as follows:

- Parallelizing existing 32-bit sequential random number generator code.
- Implementation of 64-bit sequential random number generator.
- Parallelizing the 64-bit sequential random number generator code.

These random number generators would be combined LCG's, which should make for better randomness properties. This work will also include determining values for A, C and

m for 64-bit LCGs that would give optimal period and then testing these generators to observe their randomness properties using the spectral test, so we get a densely packed even spread as in Figure 1. We expect that on completion of the work we will have all the generators working properly and meeting all the conditions for an acceptable generator as stated above.

References

[1] D.E. Knuth, *Seminumerical Algorithms*, 2nd Ed., Vol. 2 of *The Art of Computer Programming*, Addison-Wesley, Reading PA, 1981.

[2] Stephen K. Parker & Keith W. Miller, *Random Number Generators: Good ones are hard to find*, Communications of the ACM, Volume 31, Number 10, 1988.

[3] *Random Number Generators*, Computational Science Education Project, 1995. Available from `http://csep1.phy.ornl.gov/rn/rn.html`.

[4] Pierre L'Ecuyer, *Random Numbers for Simulation*, Communications of the ACM, Vol. 33, Number 10, 1990.

[5] P. D. Coddington, *Analysis of Random Number Generators Using Monte Carlo Simulation*, Int. J. Mod. Phys. C, Vol. 5, p. 547, 1994.

[6] Pierre L'Ecuyer & Serge Cote, *Implementing a Random Number Package with Splitting Facilities*, ACM Transactions on Mathematical Software, Vol. 17, No. 1, 1991.

[7] B.A. Wichmann & I.D. Hill, *An efficient and portable pseudorandom number generator*, Appl. Stat., Vol. 31 (1982).

[8] P.D. Coddington, S.H. Ko, W.E. Mahoney and J.M. del Rosario, *Monte Carlo Tests of Parallel Random Number Generators*, Northeast Parallel Architectures Center at Syracuse University Technical Report, in preparation.

[9] Pierre L'Ecuyer, *Efficient and Portable combined Random Number Generators*, Communications of the *ACM*, Volume 13, Number 6, 1995.