The Eighth International Symposium on Operations Research and Its Applications (ISORA'09)
Zhangjiajie, China, September 20–22, 2009
Copyright © 2009 ORSC & APORC, pp. 448–452

# An Implementation of a 5-term GFSR Random Number Generator for Parallel Computations

Hajime Miyazawa* Masanori Fushimi†

Faculty of Information Sciences and Engineering, Nanzan University
27 Seirei-cho, Seto, Aichi 489-0863, Japan

**Abstract** This paper describes an implementation of a 5-term GFSR (Generalized Feedback Shift Register) random number generator that generates mutually uncorrelated random number sequences on computing nodes of networked personal computers (PCs). As GFSR generators have extremely long periods and their autocorrelation functions are known, it is possible for the generator on each computing node to generate a subsequence uncorrelated with each other if initial terms of the generator are properly set up on each computing node. Some preliminary results of simple Monte Carlo simulations are also shown.

**Keywords** Random Number Generator; 5-term GFSR; Parallel Computations

## 1 Introduction

Recent advance in multi-core CPUs makes it possible to use a commodity PC for parallel computations. Even a low-cost PC can have a quad-core CPU to have four threads of execution. We can easily construct large-scale parallel computing environment by connecting a few of such PCs via Ethernet LAN.

Such parallel computing environment is suitable especially for large-scale Monte Carlo simulations because they need lots of computational time. Assigning replications in a simulation onto each computing node and collecting the computational results of the nodes enables obtaining the final answer more quickly as the number of computing node becomes larger.

One of the problems in running large-scale Monte Carlo simulations on parallel computing environment is to guarantee that the pseudorandom number generator on each computing node generates a sequence of random numbers uncorrelated with each other. A solution to this problem is to choose one generator whose autocorrelation function is known and to use mutually uncorrelated subsequences on the computing nodes[3].

This paper describes an implementation of a 5-term GFSR random number generator that generates mutually uncorrelated random number sequences on computing nodes of networked PCs. Because the autocorrelation functions of GFSR generators are known,

*miyazawa@nanzan-u.ac.jp
†fushimi@nanzan-u.ac.jp

we can verify there are no correlations among the sequences to be generated by the generators on computing nodes. Extremely long periods of the generators guarantee that they generate long sequences enough for practical use.

The rest of this paper is structured as follows. Section 2 describes the characteristics of GFSR generators. We show how the generators are used for parallel computations in Section 3. Our implementation of the pseudorandom number generator used for parallel computations is described in Section 4. Section 5 gives some results of computational experiments using our library. Section 6 concludes the paper.

# 2 Characteristics of GFSR Generators

A sequence of integers $\{X_t\}$ generated by a GFSR generator is defined recursively as follows:

$$X_{t+p} = X_{t+q_1} \oplus X_{t+q_2} \oplus \cdots \oplus X_{t+q_\ell} \oplus X_t \tag{1}$$

Here, the symbol $\oplus$ means the bit-wise addition modulo 2, and the characteristic polynomial of the recursion (1)

$$f(z) = z^p - z^{q_1} - z^{q_2} - \cdots - z^{q_\ell} - 1 \tag{2}$$

is a primitive polynomial of degree $p$ over GF(2). In order to generate a sequence of pseudorandom numbers, we must give initial values $X_1, X_2, \cdots, X_p$. The appendix of [1] describes an initialization method based on the method proposed by Fushimi[2]. This method guarantees a good autocorrelation property as well as a good multidimensional property. The autocorrelation function of the sequence generated with this initialization method is given below as follows[2].

Let $\{X_t\}$ be the $b$-bit integer sequence generated by the recursion (1) and $\{x_t\} = \{X_t/2^b\}$ be the normalized sequence. The periods of these sequences are $T = 2^p - 1$. Let $\bar{x}$ denote the average of $x_t$'s over the entire period, which is equal to $0.5(1-2^{-b})/(1-2^{-p})$ and very close to 0.5 if $p$ is large. Let $R(s)$ be the autocorrelation function of the sequence $\{x_t\}$ defined by the following:

$$R(s) = \frac{1}{T}\sum_{t=1}^{T}(x_t - \bar{x})(x_{t+s} - \bar{x}) \tag{3}$$

It is shown in [2] that

$$R(s) = -\frac{1}{4T}(1-2^{-b})^2 \qquad (1 \le |s| \le s_0), \tag{4}$$

which is almost equal to zero because $T$ is very large. Here, $s_0 = (T+1)/b$ if $b$ is a power of 2, e.g. $b = 32$, and in general $s_0 = (T+1)/b'$, where $b'$ is the smallest power of 2 which is not less than $b$. Thus $s_0$ is also very large if, for example, $p = 521$ and $b = 32$.

# 3 Using GFSR Generators for Parallel Computation

In order to ensure the independence of generated sequences, we use one and the same generator for a simulation, and use mutually uncorrelated subsequences on computing nodes. More precisely, we choose an integer $\tau$ which is less than or equal to $s_0/m$, where

$m$ is the number of computing nodes, and use the subsequences $\{X_t : (k-1)\tau+1 \le t \le k\tau\}$ on the $k$-th computing node ($k = 1,2,\cdots,m$). Then we must compute "initial values" for computing nodes no.2 through no.$m$ from the initial values $\boldsymbol{X}_0^{(1)} = \{X_1, X_2, \cdots, X_p\}$ for the computing node no.1. To compute these initial values quickly when $\boldsymbol{X}_0^{(1)}$ is given, we make preparations for initialization before $\boldsymbol{X}_0^{(1)}$ is specified.

It is known that the sequence $\{X_t\}$ which is generated by the recursion (1) satisfies

$$X_t = X_{t-e(p-q_1)} \oplus X_{t-e(p-q_2)} \oplus \cdots \oplus X_{t-ep} \tag{5}$$

where $e$ is any integral power of 2[3]. Using the recursion (5), we can compute $X_t$ from $\boldsymbol{X}_0^{(1)}$ rather quickly even for a very large $t$. Let $g(u)$ denote the greatest integral power of 2 which does not exceed $u$. Let $e = g(t/p)$ in (5), then $X_t$ can be computed from $X_{t'}$'s on the right-hand side with relatively small indices $t'$. We apply the same technique to those $X_{t'}$'s, and repeat this process until $X_t$ is expressed as a linear combination (in the sense of $\oplus$) of the elements of $\boldsymbol{X}_0^{(1)}$. This procedure can easily be implemented if we use a programming language which provides a function of recursive calls, e.g. C. The computational time required is, however, rather long when $t$ is very large and the number of terms in the recursion (5) is not very small. So we modify the procedure for very large $t$ to speed up the initialization process, which will be described in the next section for the recursion with 5 terms.

# 4 Implementation of a GFSR Generator for Parallel Computation

We implemented a 5-term GFSR generator shown in Section 3 as a library used in parallel computation environment. The implemented generator is based on the recursion

$$X_{t+521} = X_{t+86} \oplus X_{t+197} \oplus X_{t+447} \oplus X_t \tag{6}$$

First, we use the recursive procedure described above to express $X_{8193}$ as a linear combination of the elements of $\boldsymbol{X}_0^{(1)} = \{X_1, X_2, \cdots, X_{521}\}$. Using this expression, it is easy to express $X_{8194}, X_{8195}, \cdots, X_{8713}$ as linear combinations of the elements of $\boldsymbol{X}_0^{(1)}$. Thus we obtain the matrix $A$ which expresses the transformation

$$\boldsymbol{Y} = A\boldsymbol{X}_0^{(1)}$$

where $\boldsymbol{X}_0^{(1)}$ and $\boldsymbol{Y} = \{X_{8193}, X_{8194}, \cdots, X_{8713}\}$ are understood to denote column vectors. Once the matrix $A$ is determined, repetitive squarings of the matrix generate ones to compute terms much more distant from the initial terms. Finally we obtain the matrix with which we can compute $X_n$ for $n = 2^{500}$. The final matrix is incorporated in the library as a constant to compute sets of initial values for all the computing nodes from the initial values $\boldsymbol{X}_0^{(1)}$ for the first node. On the $k$-th computing node the generator generates a sequence of pseudorandom numbers beginning from the term $2^{500}$ apart from that on the $(k-1)$-th node. As the period of the entire sequence is $2^{521}-1$, more than $2^{20} \approx 1,000,000$ computing nodes generate mutually uncorrelated random number sequences.

As a parallel communication library, we adopt the OpenMPI library, which is a typical implementation of Message-Passing Interface (MPI).

# 5 Computational Experiments

Our experimental environment consists of four PCs each of which has a 3GHz AMD Phenom II X4 940 processor and 6GBytes of memory running Debian Linux with kernel version 2.6.24-amd64. They are connected via 1000Base-T Ethernet LAN. Because a Phenom II processor has four cores of computation, we can use up to 16 cores in our experiments.

## 5.1 Computing the Value of $\pi$

We ran a simple Monte Carlo simulation to compute the value of $\pi$. We compared the three generators each of which generates $2^{40}$ points in the unit square (0.0, 0.0) - (1.0, 1.0) on all computing nodes. Tables 1 and 2 show the results of the simulations on a single computational core and on 16 computational cores respectively. In the case of Table 2, the total number of sample points is $16 \times 2^{40} = 2^{44}$.

The three compared generators are our GFSR generator (global GFSR), the same GFSR but with a different random number seed on each computing node (local GFSR) and the random number generator in the C library functions(`random(3)` [1]), whose random number seed is different on each computing node.

Table 1: The values of $\pi$ obtained in the simulations (1 core)

| Generator | The value of $\pi$ | Error $\times 10^{10}$ | Simulation time (sec.) |
|---|---|---|---|
| Global GFSR | 3.1415930641 | 4105 | 21486 |
| Local GFSR | 3.1415930641 | 4105 | 21524 |
| `random(3)` | 3.1415932465 | 5925 | 39531 |
| The true value of $\pi$ | 3.1415926536 | | |

Table 2: The values of $\pi$ obtained in the simulations (16 cores)

| Generator | The value of $\pi$ | Error $\times 10^{10}$ | Simulation time (sec.) |
|---|---|---|---|
| Global GFSR | 3.1415927463 | 927 | 21949 |
| Local GFSR | 3.1415930322 | 3786 | 25500 |
| `random(3)` | 3.1415921943 | 4593 | 40483 |
| The true value of $\pi$ | 3.1415926536 | | |

We observe the following from Tables 1 and 2.

- There are no significant differences in simulation times for 1 core and 16 cores.
- There are no significant differences in simulation times for global GFSR and local GFSR, but the simulation time for `random(3)` is about twice the times for GFSRs.
- The error contained in the computed value is defined to be the absolute value of the difference between the computed value and the true value of $\pi$. Table 1 shows there is no significant differences among the errors for the three generators, but some differences are observed in Table 2: the error for the global GFSR is rather

[1] The `random(3)` function uses a non-linear additive feedback random number generator employing a default table of size 31 long integers. The period of the generator is approximately $16 \times (2^{31} - 1)$[4].

Table 3: The values of $(1/2)^5 V_5$ obtained in the simulations (16 cores)

| Generator | The value of $(1/2)^5 V_5$ | Error $\times 10^{10}$ | $D_{16}$ | $p$-value |
|---|---|---|---|---|
| Global GFSR | 0.1644932381 | 1686 | 0.340 | 0.037 |
| Local GFSR | 0.1644935708 | 1641 | 0.275 | 0.146 |
| `random(3)` | 0.1644934765 | 698 | 0.247 | 0.238 |
| The true value of $\left(\pi^2/60\right)$ | 0.1644934067 | | | |

small compared with the other two. Comparing Table 1 with Table 2, we notice that the errors for local GFSR and `random(3)` do not differ very much between the two tables, but the error for global GFSR in Table 2 is about a quarter of its counterpart in Table 1. We remember the general theory in Monte Carlo simulation that the expected error contained in the computed value obtained from i.i.d. samples is inversely proportional to the square root of the sample size. The sample size in Table 2 is 16 times the sample size in Table 1, $(1/\sqrt{16}) = (1/4)$, and this ratio is consistent with the general theory. On the other hand, the errors for local GFSR and `random(3)` in Tables 1 and 2 are rather inconsistent with the general theory.

### 5.2 Multidimensional Behavior

We also ran another Monte Carlo simulation to compute the volume of the five-dimensional unit hypersphere $V_5$. We compared the three generators each of which generates $2^{40}$ points in the five-dimensional unit hypercube on all computing nodes. Table 3 shows the result of the simulation on 16 computational cores, which actually compute $(1/2)^5 V_5 = \left(\pi^2/60\right)$. In this case, there is no significant difference among the errors for the three generators. We have also checked the distribution of the approximate values obtained on the 16 cores, and computed two-sided Kolmogorov-Smirnov statistics $D_{16}$ and the corresponding $p$-values. The null hypothesis is "the 16 values are random samples from a normal distribution," and the alternative hypothesis is two-sided.

## 6 Conclusion

We have shown a method to use GFSR pseudorandom numbers for parallel computations. A practical implementation of the method which generates mutually uncorrelated random sequences on each node is also available. Some preliminary computational experiments using the library are presented.

## References

[1] Japanese Industrial Standards Committee. *JIS Z9031:2001 Procedure for Random Number Genration and Randomization*. Japanese Industrial Standards Association, Tokyo, 2001. in Japanese.

[2] Masanori Fushimi. *Random Numbers*. University of Toyo Press, Tokyo, 1989. in Japanese.

[3] Masanori Fushimi. Random numbers for parallel computations. In Shigeyoshi Ogawa, editor, *RIMS Reports of The 7th Workshop on Stochastic Numerics*, RIMS Kôkyûroku No.1462, pages 57–62. Research Institute for Mathematical Sciences, Kyoto University, January 2006.

[4] GNU. *Linux Programmer's Manual* `random(3)`, August 2000.