The Equipartition Method for Parallel Generation of Random Numbers

Jun Makino
Department of Management Information,
Fukuyama Heisei University, Fukuyama 720, Japan

Abstract

Random numbers in a typical parallel computation can be considered to form a two dimensional array whose rows are generated in parallel. A simple and general method of constructing such an array is the equipartition method. In this method, a period of a random number sequence is divided into segments of equal length which are then arranged in rows or in columns to form an array. The row-wise arrangement is called a horizontal configuration and the column-wise one a vertical configuration.

Parallelization in the equipartition method is useful only if both the initialization and the generation procedures can be executed efficiently. We show that generators of the lagged-Fibonacci type — additive, subtractive, multiplicative generators as well as shift register generators — can be parallelized in horizontal configurations. For the shift register method, parallelization in vertical configurations is also available.

In order to apply the equipartition method in a horizontal configuration, one must evaluate terms hugely separated from the initial terms in the original sequence. For lagged-Fibonacci generators, algorithms based on a single prescription permit to compute remote terms. Assuming that the degree of the recurrence is p, the complexity of the algorithms to fix the nth term is $O(p \log n)$ for shift register generators, and is $O(p^2 \log n)$ for additive, subtractive and multiplicative generators.

In many parallel Monte Carlo computations, randomness of the array both in the row and in the column directions plays important roles. In the equipartition method, the array has a rather trivial structure when it is viewed as an arrangement of the equipartitioned strings. The structure in the orthogonal direction is, however, not trivial at all. Our discussion shows that we can tune the equipartitioning so that all the orthogonal strings as a whole form a periodical sequence with the same period as the original sequence.

For shift register generators, the parallel structure can be analyzed in more detail. For this purpose, we introduce a formalism named the PFSR. In designing a parallel shift register generator, one must take care that no duplication occurs not only in the sequences of random numbers but also in the shift register sequences generated in parallel bit positions. The PFSR formalism gives simple criteria for judging whether these conditions are satisfied or not.

Contents

1 Introduction **3**

2 Generation of Shift Register Random Numbers on Vector Processors **7**

2.1 Equipartition method on vector processors 7

2.2 Vectorial generation of shift register random numbers 8

2.3 Initialization of random vectors 11

2.4 Some comments on the vectorial generator 14

3 Generation of Shift Register Random Numbers on Distributed Memory Multiprocessors **16**

3.1 Programming style of distributed memory multiprocessors 16

3.2 Random number generation on multiprocessors 17

3.3 Sample generator 20

4 Lagged-Fibonacci Random Number Generators on Parallel Computers **24**

4.1 Equipartition method for lagged-Fibonacci sequences 24

4.2 Lagged-Fibonacci generators 25

4.3 Parallelization 26

4.4 Large delay for $F(p, q, \pm)$ 27

4.5 Large delay for $F(p, q, *)$ 30

4.6 Implementation 31

5 Binary Method of Evaluating Remote Terms in a Recurrent Sequence **36**

5.1 General prescription 36

5.2 Shift register generators 38

5.3 Additive and subtractive generators 40
5.4 Multiplicative generators 41
5.5 Comparison to existing algorithms 43

6 Structure of Parallelized Random Number Sources 45
6.1 Parallelized random number source 45
6.2 General theory 46
6.3 Applications 52

7 Parallelized Feedback Shift Register Generators 55
7.1 FSR sequences 55
7.2 GFSR generators 57
7.3 PFSR generators 58
7.4 Phase shift analysis of the PFSR generators 60
7.5 Examples 62

8 Conclusion 69

Chapter 1

Introduction

The Monte Carlo method has become a powerful and indispensable approach in many branches of science. It is characterized by the use of random numbers generated on computers. Study of pseudorandom number generation has a long history tracing back almost to the birth of electronic computers [14]. The main purpose is of course to get a good source of random numbers. Here a random number generator may be said to be good if it fulfils three criteria: a long period, high speed and randomness. The congruential method and the lagged-Fibonacci method have been used as good generators which satisfy these criteria to some extent. But the advent of supercomputers added new aspects to each of these three conditions.

First, a random number generator is now required to have a period much longer than ever: Large scale Monte Carlo simulations are the main task of supercomputers, and they use a huge amount of random numbers. In fact, 2^{30} random numbers, which constitute the whole period of a typical congruential sequence, can be generated within a few seconds on many of these machines.

Secondly, high speed generation of random numbers on a supercomputer requires an algorithm especially suited to the machine. The high performance of a supercomputer is based on parallel processing in which multiple parts of a computation are concurrently executed. The parallelism exploited by a supercomputer is deeply related to the architecture of the machine, and a programer must carefully select algorithms to extract full performance of the machine. A parallel algorithm of random numbers is hoped to achieve efficiency not only of the generation itself but also of the total computation in which random numbers are consumed.

Lastly, randomness is required in an extended meaning on supercomputers [16]. Random numbers generated on computers are not random in the strict sense, because they are generated by a definite algorithm. They merely seem to be random for most applications, if they are produced by a good generator which would pass a series of standard statistical tests [14]. For parallel computations, the situation is much more complex. Though random numbers form a single sequence in a traditional computation, random vectors are produced serially on vector processors and multiple sequences are generated in parallel on multiprocessors. Thus random numbers form a two dimensional array of numbers in a typical parallel computation. The new feature is that randomness is required not only for each vector or each sequence but also for the two dimensional array of random numbers. We have little knowledge about such randomness, and randomness is presumably the most serious problem for parallel generation.

The purpose of the present work is to develop a parallel method of generating random numbers taking into account the three criteria described above. We call the method the equipartition method because it is based on equipartitioning of a single random number sequence. We apply the method to parallelize lagged-Fibonacci sequences, whose periods are so long that sufficient amount of numbers can be generated for any conceivable scale of computation. In this method, random numbers are produced quite efficiently: maximum improvement of speed is acquired by the parallelism. Furthermore, the equipartition method gives the two dimensional array a definite structure which enables us to analyze the parallel generator theoretically.

The importance of theoretical analyzability is indicated by the prevailing confidence in the congruential method. Though a lot of methods have been proposed for producing random numbers on computers, the congruential method has been the most popular one all the time. Indeed the congruential method may be quite simple and very efficient, but it has a well-known drawback: Points in a high-dimensional space concentrate in a small number of parallel hyperplanes [22]. The very reason for the popularity of congruential generators seems to be that their theoretical properties are well understood including such a defect. The shift register method, which is increasing its popularity in recent years, is the method whose theoretical analysis is most advanced next to the congruential method. In designing a parallel generator, we have to fulfil the criterion that theoretical analysis of its properties is feasible. When sequences are generated in parallel, randomness is required not only within

each sequence but over whole the construct made up by the sequences. If generators of quite different types were used in parallel, the analysis of such a construct would be very difficult. In this respect, parallelization based on equipartitioning of a single sequence seems to be especially favorable.

In the equipartition method, a string of random numbers which forms a period of a pseudorandom sequence is divided into segments with equal length. These segments are then arranged to form a two-dimensional array, the rows of which can be generated and utilized in parallel. (The direction in which random numbers are sequentially generated will be taken to be the row direction.) On a vector processor, for example, column-vectors of such an array are generated sequentially on a vector register. On multiprocessors, each row can be assigned to each node processor. Such an array will be denoted as $(v_{i,j})$ and will be called a parallelized random number source or a parallelized source for short. In utilizing a pseudorandom sequence, one should not generate more than a small part of the total period. In this respect, a string of random numbers which forms a period in the sequence may be called a random number source. In applying a parallelized source, which is a two-dimensional arrangement of such a source, one must take care that the generated numbers are only a small portion of the parallelized source. Two different types of arrangement are used to construct a parallelized source from a given sequence. One of these is a horizontal configuration in which a random number source from the original sequence is divided into segments of equal length which are then arranged row by row to construct a parallelized source. The other is a vertical configuration where segments are arranged column by column. Both of these configurations are widely applied in parallel computations.

The work consists of two parts. From chapter 2 to chapter 5, we describe how to parallelize lagged-Fibonacci generators, presenting various algorithms necessary for the parallelization. In chapters 6 and 7, we discuss properties of the parallelized random numbers through theoretical analysis of the structure of parallelized sources.

Among the lagged-Fibonacci generators, shift register generators have much simpler structure than the other generators. Because we can derive more results for shift register generators, we treat these generators separately from the other generators. For example, a shift register generator is easily parallelized not only in a horizontal configuration but in a vertical configuration. Furthermore, parallelization of a shift register generator involves

many interesting features specific to vectorial computers. Therefore, we discuss generation of shift register random numbers on vector processors in the next chapter. Chapter 3 deals with the parallelization of the shift register method on distributed memory multiprocessors. Other types of the lagged-Fibonacci generators with $+$, $-$ or $*$ are parallelized in chapter 4. In chapter 5, we propose a general prescription to evaluate remote terms in a recurrent sequence. Chapter 6 is devoted for the analysis of the structure of the parallelized random number source when it is viewed in the row direction or in the column direction. The analysis of the structure is performed in more depth for parallelized shift register generators in chapter 7. Chapters 2, 3, 4, 6 and 7 are based on the author's published papers [17], [18], [20], [19] and [21] respectively.

Chapter 2

Generation of Shift Register Random Numbers on Vector Processors

2.1 Equipartition method on vector processors

A set of random numbers, which are generated vectorially with a vector length l, may be denoted by a column vector with l components. Typically, such a generation is repeated a large number of times m, resulting in a matrix of random numbers $v_{i,j}$ $(0 \leq i \leq l-1, 0 \leq j \leq m-1)$. The problem is that no general method is known to define and check the randomness of such a two dimensional array of numbers. To circumvent this problem, a single sequence $\langle x_i \rangle$ with established randomness can be used to construct $\langle v_{i,j} \rangle$ in such a way that the randomness most important for $\langle v_{i,j} \rangle$ is assured by the randomness of $\langle x_i \rangle$.

Typically, there are two types of configurations in which $\langle x_i \rangle$ is arranged into $\langle v_{i,j} \rangle$. One is to arrange $\langle x_i \rangle$ in the column direction with a constant spacing L between columns:

$$v_{i,j} = x_{i+jL}.$$

Such an arrangement will be called a vertical configuration. The other possibility is a horizontal configuration in which $\langle x_i \rangle$ is arranged in the row direction as

$$v_{i,j} = x_{iM+j}.$$

Vectorization of a single random sequence in either of these two configurations is called the equipartition method. Since randomness of the original sequence is assured only in a restricted sense, the configuration of vectorial generation must be chosen carefully to fit the randomness required in the program.

2.2 Vectorial generation of shift register random numbers

High speed generation on vector processors requires an algorithm especially suitable to these machines. Above all, the generation algorithm must be vectorizable. Furthermore, it is hoped to be used inline in the application program in which random number generation is involved. The advantage of inline random number generation was once advocated by Marsaglia and Bray [23]. But it becomes more important on vector processors, because `CALL` statements are not vectorized by vectorizing compilers. Thus a call for a subroutine may interrupt the total performance of a vectorized program, even if the subroutine itself has a satisfactory efficiency.

The shift register method of random number generation has attracted many researchers since it was first introduced by Tausworthe in 1965 [29]. The main interest has been the possibility to prevent multidimensional lattice structure found in congruential sequences [22]. In this method, a sequence of random numbers $\langle x_i \rangle$ is generated in terms of a linear recurrence of degree p,

$$x_i = x_{i-p} \oplus x_{i-q}. \tag{2.1}$$

Here p and q are integers which satisfy $0 < q < p$, and $\oplus$ denotes bitwise exclusive-or operation [15]. As is easily seen from (2.1), the reciprocal recurrence $x_i = x_{i-p} \oplus x_{i-(p-q)}$ generates the same sequence in reverse order. If we define an operator D by $x_i = Dx_{i-1}$, the recurrence (2.1) is written as a trinomial over $GF(2)$:

$$D^p + D^{p-q} + 1 = 0.$$

In terms of another operator X defined by $x_i = Xx_{i+1}$, the recurrence relation is expressed as $X^p + X^q + 1 = 0$. If the characteristic trinomial $x^p + x^q + 1$ is primitive over $GF(2)$, the period of the sequence is $2^p - 1$. Lists of primitive trinomials are found in references [31, 32, 33], and one can choose p appropriately so that the period is long enough for any application.

As is seen from logical independence, the recurrence (2.1) can generate up to q consecutive terms in parallel. This parallelism is extended to $q \cdot 2^k$ terms, if one uses a relation

$$x_i = x_{i-p\cdot 2^k} \oplus x_{i-q\cdot 2^k} \tag{2.2}$$

to generate random numbers. (2.2) is an immediate consequence of (2.1) because

$$D^{p\cdot 2^k} + D^{(p-q)\cdot 2^k} + 1 = (D^p + D^{p-q} + 1)^{2^k}$$

by the squaring property of polynomials over $GF(2)$. It is possible to vectorize generation of shift register random numbers based on this parallelism. But such a generator is not appropriate for inline use, because the vectorized loop will be complicated by instructions to control the table keeping the latest series of random numbers for the next generation.

The vectorization method adopted in this chapter is to generate a sequence of random vectors by a vectorial recurrence

$$v_{i,j} = v_{i,j-p} \oplus v_{i,j-q}. \tag{2.3}$$

Thus each row of $\langle v_{i,j} \rangle$ is generated by itself in terms of the original recurrence (2.1). In this method, the logical independence required for the vectorization is obvious. If the phase of generation j is allowed to depend on i, generation of each row sequence may be made asynchronously. But hereafter j is assumed to be independent of i.

At first sight, (2.3) may seem to be applicable only to horizontal configurations. However, it can also generate a vertical configuration if the spacing between columns is of a special value. To see this, notice that each row of a vertical configuration is a subsequence consisting of every Lth term of the original sequence $\langle x_i \rangle$. If L is relatively prime to $2^p - 1$, such a subsequence is also a shift register sequence generated by a primitive polynomial of degree p. Thus the row sequences can be generated by applying a single recurrence of degree p in parallel. For general L, the recurrence includes exclusive-or of many terms so that generation of $\langle v_{i,j} \rangle$ will be time consuming. But if L is chosen to be an integer power of 2, the recurrence coincides with (2.3): In a vertical configuration

$$v_{i,j} = x_{i+j\cdot 2^\mu}, \tag{2.4}$$

the vectorial recurrence (2.3) is an immediate consequence of (2.2). In this configuration, every rows as well as columns of $\langle v_{i,j} \rangle$ constitute consecutive parts of the original sequence $\langle x_i \rangle$. But the assured randomness of the original sequence is reflected only in column sequences, because randomness of $\langle x_i \rangle$ is usually examined for only a small part of the long period in the neighborhood of x_0.

The great advantage of (2.3) is that it can be used inline in FORTRAN programs. In actual codes, a two dimensional array of integers, say IRAND, is used rotationally to store p consecutive vectors. First of all, initial values have to be given to all $l \times p$ elements of IRAND. (A simple method to do this will be given in the next section.) Then, initializing J1 and K1 to -1 and $p-q-1$ respectively, one can generate vectors of random numbers in such a code as

```
      DO 200 N=0,m-1
        J1=MOD(J1+1,p)
        K1=MOD(K1+1,p)
        DO 100 I=0,l-1
          IRAND(I,J1)=IEOR(IRAND(I,J1),IRAND(I,K1))
100     CONTINUE
200   CONTINUE
```

where IEOR denotes bitwise exclusive-or. The deepest loop is vectorized, and it actually contains many instructions which may form the main step of a Monte Carlo simulation.

Practically, several random vectors may be generated in a vectorized loop. It is quite easy to generalize the above code to these cases. For example, in order to generate random vectors twice in the vectorized loop, introduce a set of new variables J2 and K2 with initial values 0 and q respectively, and insert a new line

```
IRAND(I,J2)=IEOR(IRAND(I,J2),IRAND(I,K2))
```

in the deepest vectorized loop. In the outer loop, J1,K1,J2,K2 should be added by two modulo p.

The performance of the algorithm would be appreciated when it is applied inline in simulation programs. But for reference, the speed of generation was tested by the code described above with $p = 250$ and $q = 103$. On SX-2N, the time per one number generation was 5.5 nsec when $l = 256$ which is the saturation length of this machine. The corresponding value on S820/80 was 3.6 nsec, but the speed on this machine was higher for a larger vector length. For example, it was 1.7 nsec for $l = 16384$.

2.3 Initialization of random vectors

For a recurrence of degree p to define a sequence, p initial values must be given for the sequence. Thus the shift register recurrence (2.1) needs values of $x_0, x_1, \ldots, x_{p-1}$ to define the whole sequence $\langle x_i \rangle$. Analogously, the vectorial recurrence (2.3) requires p initial vectors to be specified: All $l \times p$ values of `IRAND` or equivalently $v_{i,j}$ $(0 \le i \le l-1, 0 \le j \le p-1)$ must be initialized. To construct $\langle v_{i,j} \rangle$ from $\langle x_i \rangle$, two initialization procedures are necessary to define $\langle x_i \rangle$ and then to define $\langle v_{i,j} \rangle$. Various methods to initialize $\langle x_i \rangle$ are seen in the literature [15, 13, 6]. In this section, a simple procedure is given to initialize $\langle v_{i,j} \rangle$ assuming that $\langle x_i \rangle$ has been initialized by an appropriate method. One of the problems in initializing $\langle v_{i,j} \rangle$ is that the number of values to be fixed will be considerably large. But this will not be a point if the initialization procedure is vectorizable. Another problem is quite serious for a horizontal configuration. Because the spacing M between rows are generally very large, the set of initial values for $\langle v_{i,j} \rangle$ involves numbers distantly separated in $\langle x_i \rangle$. As it is impracticable to calculate all the numbers up to $x_{(l-1)M+p-1}$ recursively, an especially efficient algorithm is needed to initialize a horizontal configuration.

To get a simple initialization procedure for a horizontal configuration, let the spacing be restricted to be a power of 2 so that

$$v_{i,j} = x_{i \cdot 2^\mu + j}. \tag{2.5}$$

A large delay can be evaluated by dividing operators such as $D^{i \cdot 2^\mu}$ by $D^p + D^{p-q} + 1$ to obtain the remainder polynomial [2]. In fact, if all the coefficients $c_{i,j}$ are fixed (to be 0 or 1) in

$$D^{i \cdot 2^\mu} = c_{i,0} + c_{i,1} D + \cdots + c_{i,p-1} D^{p-1}, \tag{2.6}$$

each $v_{i,j}$ in (2.5) is given by

$$v_{i,j} = c_{i,0} x_j \oplus c_{i,1} x_{j+1} \oplus \cdots \oplus c_{i,p-1} x_{j+p-1}. \tag{2.7}$$

Thus each of the values to be initialized for $\langle v_{i,j} \rangle$ can be evaluated as a linear combination of p consecutive terms in $\langle x_i \rangle$ near the origin of the sequence x_0.

The computation of $\langle c_{i,j} \rangle$ can be performed efficiently by using the squaring property of polynomials over $GF(2)$ [30]. First let $l_0 = \min(l, p)$, and set $c_{i,j} = \delta_{i,j}$ $(0 \le i \le l_0 - 1, 0 \le j \le p-1)$. Then the following code rewrites `C(I, J)` to the coefficients in (2.6).

```
      DO 500 K=μ − 1,0,-1
        DO 310 J=p − 1,0,-1
          DO 300 I=0,l0 − 1
            C(I,2*J+1)=0
            C(I,2*J)=C(I,J)
300       CONTINUE
310     CONTINUE
        DO 410 J=2p − 2,p,-1
          DO 400 I=0,l0 − 1
            C(I,J−p)=IEOR(C(I,J−p),C(I,J))
            C(I,J−q)=IEOR(C(I,J−q),C(I,J))
400       CONTINUE
410     CONTINUE
500   CONTINUE
```

The innermost `DO` loops with terminals `300` and `400` do the work vectorially for all i's. To see the idea behind the above code, ignore these loops and consider a fixed i for a moment. When the `DO` loop `310` starts, $D^{i\cdot 2^{\mu}}$ is expressed as

$$c_{i,0} + c_{i,1} D^{1\cdot 2^{k+1}} + \cdots + c_{i,p-1} D^{(p-1)\cdot 2^{k+1}},$$

which is the expansion in the basis $\{1, D^{1\cdot 2^{k+1}}, \cdots, D^{(p-1)\cdot 2^{k+1}}\}$. Observing that the basis can be written as $\{1, D^{2\cdot 2^{k}}, \cdots, D^{(2p-2)\cdot 2^{k}}\}$, the loop `310` shifts the coefficients to give an extended expression

$$c_{i,0} + c_{i,1} D^{1\cdot 2^{k}} + \cdots + c_{i,j} D^{j\cdot 2^{k}}$$

with $j = 2p-2$. The length $j+1$ in this expression can be reduced by one, by rewriting the last term using a relation $D^{j\cdot 2^{k}} = D^{(j-p)\cdot 2^{k}} + D^{(j-q)\cdot 2^{k}}$ which is equivalent to (2.2). The `DO` loop with terminal `410` repeats this until the expression becomes

$$c_{i,0} + c_{i,1} D^{1\cdot 2^{k}} + \cdots + c_{i,p-1} D^{(p-1)\cdot 2^{k}},$$

which is the expansion in the basis $\{1, D^{1\cdot 2^{k}}, \cdots, D^{(p-1)\cdot 2^{k}}\}$. In this way, the loop `500` rewrites the coefficients $\langle c_{i,j} \rangle$ repeatedly as the basis changes down to the canonical one. Notice that the simplicity of the algorithm is a result of the restricted spacing in (2.5).

Now the whole procedure to initialize a horizontal configuration (2.5) can be described. Assume that $x_0, \ldots, x_{p-1}$ are initialized by an appropriate method so that $\langle x_i \rangle$ has good randomness. Then the following steps initialize $\langle v_{i,j} \rangle$ properly for (2.5).

Step 1. Compute $x_p, \ldots, x_{2p-2}$.

Step 2. Compute $c_{i,j}$ $(0 \le i \le l_0 - 1, 0 \le j \le p-1)$ by the algorithm described above.

Step 3. Compute $v_{i,j}$ $(0 \le i \le l_0 - 1, 0 \le j \le p-1)$ by (2.7)

Step 4. If $p < l$, compute $v_{i,j}$ $(p \le i \le l-1, 0 \le j \le p-1)$ recursively by

$$v_{i,j} = v_{i-p,j} \oplus v_{i-q,j}. \tag{2.8}$$

The main steps 2 and 3 are necessary only in initializing the first p rows of $\langle v_{i,j} \rangle$, because the other rows are then simply given by (2.8) which is also a result of the restricted horizontal configuration (2.5).

The initialization procedure described above is very effective, and it is desirable that the same algorithm can be applied also to vertical configurations. For a vertical configuration, the necessary and sufficient condition for the vectorial recurrence (2.3) to be applied is that it is in the special form (2.4). And in this case, almost the same procedure can be applied as in the case of horizontal configurations:

Step 1. Compute $x_p, \ldots, x_{p+l_0-2}$.

Step 2. Compute $c_{i,j}$ $(0 \le i \le p-1, 0 \le j \le p-1)$.

Step 3. Compute $v_{i,j}$ $(0 \le i \le l_0 - 1, 0 \le j \le p-1)$ by

$$v_{i,j} = c_{j,0}x_i \oplus c_{j,1}x_{i+1} \oplus \cdots \oplus c_{j,p-1}x_{i+p-1}.$$

Step 4. If $p < l$, compute $v_{i,j}$ $(p \le i \le l-1, 0 \le j \le p-1)$ by (2.8).

For vertical configurations, (2.8) is a trivial relation. Notice that each step except Step 4 has a slight difference from the corresponding step for horizontal configurations. Step 2 can be coded as in the horizontal case but with the loops `300` and `400` ranging from `I=0` to $p-1$.

The vectorizability of Step 2 has already been discussed. It is easy to see that Step 3, which is another important step, is also vectorizable. Step 4 can be vectorized with respect to j. Step 1 is also vectorizable via the inherent parallelism of the recurrence described at the top of §2, but it is generally not important because the step is only a small part of the whole procedure.

The initialization procedure described in this section was coded into two `FORTRAN` subroutines, one for horizontal configurations and the other for vertical configurations. The execution times of these subroutines were short enough so that they would be negligible for large scale computations. For example, a call for a subroutine to initialize a horizontal configuration with $p = 250$, $q = 103$, $l = 256$, $\mu = 32$ needed only 0.15 sec on SX-2N and 0.066 sec on S820/80.

2.4 Some comments on the vectorial generator

The congruential method has been the most common random number generator even on vector processors. Though its period is relatively short, the simplicity and the versatility of the algorithm are the very characters required for a generator on vector processors. In this chapter, it has been shown that these properties can be shared by the shift register method: Inline generation is possible by vectorial application of the recurrence, leading to either vertical or horizontal configuration at one's disposal. Initialization for the vectorial recurrence may have been an obstacle in applying such a method, but a simple procedure is shown to do this quite efficiently for either of the configurations.

One of the penalties of the shift register method is the necessity for a large table to keep consecutive random numbers for its recurrence. The generation method discussed in this chapter prefers the vector length to be not very large. When the vector length is extraordinarily large, it may cause some trouble in allocating a large memory space for the rotation table `IRAND`. In such a case, use of a subroutine based on the parallelism inherent in the recurrence may be more appropriate [11].

Randomness of $\langle v_{i,j} \rangle$ as a two dimensional array has not been discussed. In fact, there seems to be no general way to check such randomness. In this chapter, instead, the generator is required to cover two types of configurations. For a simulation in which randomness within a row is respected, generation in a horizontal configuration may be desirable. If randomness

within columns is more important, a vertical configuration will be preferred. For many computations, however, randomness in both directions would be important. In such a case, repeated computations in both configurations will be helpful to see if the results do not depend on the sources of random numbers. It is very easy to do this because one must only change to call another subroutine to initialize $\langle v_{i,j} \rangle$.

Chapter 3

Generation of Shift Register Random Numbers on Distributed Memory Multiprocessors

3.1 Programming style of distributed memory multiprocessors

The high performance of supercomputers is based on parallel processing in which multiple parts of a computation are concurrently executed. The parallelism exploited by a supercomputer is deeply related to the architecture of the machine, and a programer must carefully select algorithms to extract full performance of the machine. Among several different architectures used on supercomputers, distributed memory multiprocessors often need specific algorithms. On these machines, a separate program runs on each processor communicating by message-passing operations. Typically, these programs are copies of a single program, and this style of programming is referred to as single code multiple data (SCMD) or single program multiple data (SPMD) [10]. In this chapter, random number generation on distributed memory multiprocessors is discussed on the assumption of this programming style.

There exist vast class of problems which are intrinsically suited to be concurrently processed. Existing random number generators, however, seem to be not of this type: A random number generator is usually defined in terms of a recurrence relation, which seems to characterize the sequential nature of the algorithm. This is not true, however, because each processor can generate random numbers by itself if the processor is given a generator for its use. Generators distributed to the processors can be based on a single algorithm with

some of the parameters different from generator to generator so that they work in parallel on multiprocessors. Today, random number generation on multiprocessors seems to be realized usually in this style. However, if the parameters are distributed to the generators carelessly, strong correlations may well occur among the generated sequences. To avoid this, a consistent method must be investigated to insure independence among the multiple sequences.

In this context, we discussed generation of shift register random numbers on vector processors in chapter 2. The method proposed there was typically to generate multiple regularly delayed versions of a single shift register random number sequence. By introducing a definite mathematical structure in the totality of multiple sequences, the method permits arranging various parameters to give the structure some desirable properties. Especially, if the delay between the neighboring sequences is taken to be a power of two, numbers generated within a random vector form a part of a shift register random number sequence that obeys the same recurrence as the original sequence. In this case, the initialization procedure necessary for the vectorial generator can be accomplished quite efficiently in vectorial fashion.

The main purpose of the present chapter is to show that the method is also available on distributed memory multiprocessors in a quite natural and efficient fashion. Note that the programming style of these computers is quite different from that of vector processors, and a vectorial generator of random numbers is not always applicable to such machines. For example, there is another way to generate shift register random numbers on a vector processor: Because a shift register recurrence usually involves only a few terms, many consecutive numbers in the sequence can be generated concurrently. (See for example [11].) But this kind of vectorial generator would be hard to implement on distributed memory multiprocessors.

3.2 Random number generation on multiprocessors

A random number generator which is extensively used on customary processors does not necessarily work well on machines of a newer type. Like many other algorithms, a generator on such a computer have to satisfy many conditions which are intimately related to the architecture and the programming style employed by the machine. So it would be appropriate to start the discussion enumerating some of these conditions required for a random number generator on distributed memory multiprocessors: (a) According to the SCMD programming

style on distributed memory multiprocessors, a single algorithm should generate multiple sequences on multiprocessors in parallel. (b) The generator is desired to work independently within each processor and to be free from communication with other processors. Then it can be applied to a parallel computation without interrupting the parallelism of the whole program. (c) Multiple sequences generated on multiprocessors are hoped to be reproducible on a traditional computer as well as on multiprocessors. This condition, together with the above one, will be helpful in debugging parallel computations which involve random number generation. (d) The multiple sequences generated on multiprocessors must be 'independent' of each other in some sense, though they could not be truly independent as they are generated by a definite algorithm.

It would be very difficult to devise a genuine parallel generator which satisfies all of these conditions. Instead, a simple approach is used where multiple parts of a single sequence separated by a long enough distance is used as multiple sequences. Thus let $\langle x_i \rangle$ be a pseudorandom sequence generated by a definite recurrence relation. The multiple sequences used on multiprocessors may be denoted as a matrix $\langle v_{i,j} \rangle$ where the i-th row is the i-th sequence. The idea is simply to let $v_{i,j} = x_{iM+j}$. The phase difference M between neighboring sequences is a constant which is chosen to be greater than the number of random numbers used in a sequence. It is evident that each row sequence is generated by the same algorithm that generates the original sequence $\langle x_i \rangle$, provided initial value(s) for each row sequence have been calculated. One advantage of this approach is that the two dimensional structure of the multiple sequences can be analyzed to some extent which may give insights to the study of 'independence' among the sequences. For example, columns of $\langle v_{i,j} \rangle$ are M-th decimation of $\langle x_i \rangle$ and would obey a definite recurrence. By examining this recurrence, it is possible to investigate properties of column sequences which are crucial for many parallel computations.

The congruential method is an example of the generator which is widely used on multiprocessors in the style discussed above. Typically, it generates a sequence of integers $\langle x_i \rangle$ by a recurrence $x_i = ax_{i-1} \bmod m$. On multiprocessors, each sequence is generated by

$$v_{i,j} = av_{i,j-1} \bmod m, \tag{3.1}$$

provided the initial value is properly set to be

$$v_{i,0} = x_{iM}.$$

A column sequence is also a congruential sequence generated by $v_{i,j} = bv_{i-1,j} \bmod m$, where $b = a^M \bmod m$. One of the defects of the congruential method is that its period is too short: The period of a typical multiplicative congruential generator on 32-bit computers is 2^{30} which is generated within a few seconds on many modern supercomputers. Another problem is that the period of a decimation of the original sequence is sometimes very short. This is especially troublesome when the modulus m is a power of 2. For example, if $\langle a_i \rangle$ is a congruential sequence with $m = 2^{32}$ and a period of 2^{30}, the period of the 2^δ-th decimation $\langle a_{i2^\delta} \rangle$ is only $2^{30-\delta}$. This is really a problem, because Monte Carlo computations on highly parallel multiprocessors are often effected by such decimations with no small values of δ and a congruential sequence of a shorter period is generally less random [12].

Generators based on a shift register sequence have fascinating properties in these respects. To see this, let the characteristic polynomial be $f(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_p x^p$ ($c_0 = c_p = 1$) which is primitive on $GF(2)$. The sequence in the i-th row is generated by

$$v_{i,j} = c_1 v_{i,j-1} \oplus c_2 v_{i,j-2} \oplus \cdots \oplus c_p v_{i,j-p}, \tag{3.2}$$

if initial values are set to be

$$v_{i,j} = x_{iM+j}, \quad j = 0, 1, \ldots, p-1. \tag{3.3}$$

First, its period $2^p - 1$ can be very long, which by itself justifies use of shift register random numbers in large scale simulations. Furthermore, the period is easily handled to be a prime number by taking the degree of the characteristic polynomial p to be a Mersenne exponent. Then a decimation of the original sequence is always a shift register sequence of the same period. Particularly, the columns of $\langle v_{i,j} \rangle$ are also a shift register random sequence whose fundamental features can be analyzed. For example, k-distribution can be established not only for the original sequence but also for the sequence generated in columns. Even if the period is not a prime number, the situation would be much more optimistic than in the congruential case because the period is very long.

Generation of random numbers in the shift register method can be faster than in the congruential method, because the recurrence (3.2) usually involves only one exclusive-or which is faster than a multiplication in (3.1). The main problem in applying a shift register generator to parallel computations is the initialization procedure preparing proper initial values as in (3.3) for each row sequence. When the delay M is a power of two, the initialization is

known to be executed quite efficiently using the squaring property of shift register sequences [17, 30]. On vector processors, the procedure can be executed in full performance of machines because it is almost completely vectorizable [17]. In the next section, the method is shown to apply equally well to multiprocessors by describing all programs including the initialization procedure in accordance with the conditions described at the beginning of this section.

3.3 Sample generator

An example of random number generator package is given in this section to show how the general idea described in the preceding section can be realized. Figure 3.1 is the sample package for initializing and generating a specified subsequence. It is coded in the standard FORTRAN77 language except it contains some bit operation functions. Though bit operations are served by many modern FORTRAN compilers, their expression varies from system to system. The code described here is intended to be used on a Fujitsu AP1000 multiprocessor system. Integers are assumed to be represented in 32 bits with 2's complement notation.

The program package given here is meant as a sample because the method described in the preceding section contains much arbitrariness. In fact, no definite idea was given to fix the characteristic polynomial $f(x)$, the initialization method for the original sequence $\langle x_i \rangle$, and the phase difference between neighboring sequences M. How to get a generator with good randomness by fixing the arbitrariness is a subject for future study. Here a single generator was chosen simply to make the description concrete. The generator of the original sequence adopted here is that of Fushimi and Tezuka [6, 4]. It is essentially a Tausworthe generator [29] based on a primitive trinomial $f(x) = x^{521} + x^{32} + 1$ with 31-bit word length and 32-bit delay between consecutive words. The delay between two neighboring subsequences was chosen to be $M = 2^{261}$.

`URAND` is a subroutine which gives a uniform random integer between 0 and $2^{31} - 1$ in its argument `IRND`. A random number uniformly distributed in [0,1) can be provided if `IRND` is divided by `2.0**31`. This subroutine is a description of the popular GFSR algorithm [15]. The common block stores two integers `JR` and `KR` and an array of integers `IW` which is a table of p consecutive numbers in the sequence. `JR` and `KR` remember relevant addresses in the table `IW` to extract a random number from the table and to replenish the table with a newly

```
      SUBROUTINE URAND(IRND)
C
      PARAMETER(IP=521,IQ=32)
      COMMON // JR,KR,IW(0:IP-1)
      IRND=IW(JR)
      IW(JR)=XOR(IW(JR),IW(KR))
      JR=JR+1
      IF (JR.EQ.IP) JR=0
      KR=KR+1
      IF (KR.EQ.IP) KR=0
      END
C
      SUBROUTINE CINIT (NSEQ)
C
      CALL INIT
      CALL DELAY(NSEQ,261)
      END
C
      SUBROUTINE INIT
C
      PARAMETER(IP=521,IQ=32,ISEED=1)
      COMMON // JR,KR,IW(0:IP-1)
      INTEGER IB(0:IP-1)
      IX=ISEED
      DO 10 I=0,IP-1
        IX=IX*69069
        IB(I)=ISHFT(IX,-31)
   10 CONTINUE
      JR=0
      KR=IP-IQ
      DO 30 J=0,IP-1
        IWORK=0
        DO 20 I=0,31
          IWORK=ISHFT(IWORK,1)+IB(JR)
          IB(JR)=XOR(IB(JR),IB(KR))
          JR=JR+1
          IF (JR.EQ.IP) JR=0
          KR=KR+1
          IF (KR.EQ.IP) KR=0
   20   CONTINUE
        IW(J)=ISHFT(IWORK,-1)
   30 CONTINUE
      END
C
      SUBROUTINE DELAY (LAMBDA,MU)
C
      PARAMETER(IP=521,IQ=32)
      COMMON // JR,KR,IW(0:IP-1)
      INTEGER IWK(0:2*IP-2),C(0:2*IP-1),IB(0:IP+31)
      DO 110 I=0,IP-1
        IWK(I)=IW(I)
  110 CONTINUE
      DO 120 I=IP,2*IP-2
        IWK(I)=XOR(IWK(I-IP),IWK(I-IQ))
  120 CONTINUE
      DO 210 I=0,MU-1
        IB(I)=0
  210 CONTINUE
      M=LAMBDA
      NB=MU-1
  220 CONTINUE
        IF ((M.GE.0).AND.(M.LE.IP-1)) GOTO 300
        NB=NB+1
        IB(NB)=AND(M,1)
        M=ISHFT(M,-1)
        GOTO 220
  300 DO 310 I=0,IP-1
        C(I)=0
  310 CONTINUE
      C(M)=1
      DO 340 J=NB,0,-1
        DO 320 I=IP-1,0,-1
          C(2*I+IB(J))=C(I)
          C(2*I+1-IB(J))=0
  320   CONTINUE
        DO 330 I=2*IP-1,IP,-1
          C(I-IP)=XOR(C(I-IP),C(I))
          C(I-IQ)=XOR(C(I-IQ),C(I))
  330   CONTINUE
  340 CONTINUE
      DO 420 J=0,IP-1
        IWORK=0
        DO 410 I=0,IP-1
          IWORK=XOR(IWORK,C(I)*IWK(J+I))
  410   CONTINUE
        IW(J)=IWORK
  420 CONTINUE
      END
```

Figure 3.1: Sample programs in SCMD programming style to initialize and generate shift register random numbers on distributed memory multiprocessors. Random integers between 0 and $2^{31}-1$ are generated by `URAND`. Preceding to the generation, `CINIT` must be called once for all.

generated random number.

Before URAND is called, subroutine CINIT must be called once for all, to initialize IW as well as JR and KR. It prepares in IW the p initial values (3.3) for the subsequence to be generated. The argument NSEQ is roughly the number for the subsequence (i in (3.2) and (3.3)) to be generated on the processor. CINIT is composed of two steps: to call INIT and then to call DELAY.

Subroutine INIT initializes p elements of the table IW so that they form initial values for the original shift register random number sequence. It also gives initial values for JR and KR. The algorithm adopted here is essentially the same as that described in [4] except the value for ISEED. It is automatically assured that the sequence is k-distributed up to $\lfloor 521/32 \rfloor = 16$ dimension.

Subroutine DELAY delays the phase of the shift register random number sequence stored in IW. Calling DELAY(λ,μ) effects each element of IW to be replaced by the number in the random sequence delayed by $\lambda 2^\mu$ terms from the original phase. (Strictly, the delay is 2^μ times the unsigned integer given by the internal 2's complement representation of λ.) The loops with labels 210 and 220 decompose the delay in a binary form

$$\lambda 2^\mu = b_\mu 2^\mu + b_{\mu+1} 2^{\mu+1} + \cdots + b_{n_b} 2^{n_b} + m 2^{n_b+1}$$

bit by bit until $0 \leq m \leq p-1$ is reached. (As n_b depends on λ, such decomposition would be an obstacle in vectorizing the initialization procedure on vector processors. In fact, the vectorial algorithm given in reference [17] does not include such decomposition.) Then through label 300 to label 340, the squaring property of shift register sequences is applied to expand the delay operator $D^{\lambda 2^\mu}$ in the canonical basis $\{1, D, D^2, \ldots, D^{p-1}\}$. Details of the algorithm has been described in [17, 30].

The program works in parallel on multiprocessors if its copies are distributed to all processors. It could be possible to call INIT on host processor and to broadcast IW to nodes. Then each node could call DELAY to get its proper initial values. But CINIT assigns INIT on each node processor. This makes the whole package free from communication programs, the virtue of which was discussed in the preceding section.

The pair of subroutines CINIT and URAND can be served as application package of random numbers on parallel processors. Users of the package must only manage to call CINIT with different arguments on different processors, in order to generate different sequences on dif-

ferent processors. For this purpose, a processor identifier `ID` given by a 'whoami' operation would be useful. For example, if `NCELL` node processors are used and each node knows its `ID` ($0 \leq$ `ID` $<$ `NCELL`), the initialization procedure can be systematically executed by a statement

$$\texttt{CALL CINIT(IB} * \texttt{NCELL} + \texttt{ID)}$$

with a definite value of `IB`. By giving various values to `IB`, one can repeat different Monte Carlo trials using non-overlapping source of random numbers.

As columns of $\langle v_{i,j} \rangle$ are 2^{261}-th decimation of $\langle x_i \rangle$, they form elements of a shift register sequence. Though the characteristic polynomial of this sequence is the same as that of the row sequence, the two sequences do not coincide by any phase-shifting. The k-distributivity of the column sequence, however, is easily checked for $1 \leq k \leq 16$ using the method described in [6].

A defect of the shift register method is that one has to keep a rather large table of consecutive random numbers. This sometimes causes problem when the present method is applied to vector processors. In fact, tables for all sequences generated in vector components must be stored in a single memory unit of definite size, and the size of the table expands in proportion to the vector length which may be specified to be very large. For distributed memory multiprocessors, the size of the local memory is usually large enough to store the table for a single sequence. It is also independent of how many nodes the system is composed of. So there seems to be little problem in memory allocation, if only a single sequence is assigned to a single processor.

Chapter 4

Lagged-Fibonacci Random Number Generators on Parallel Computers

4.1 Equipartition method for lagged-Fibonacci sequences

The random number generators that have been used conveniently on parallel machines are the linear congruential generators and the GFSR generators. Especially, the GFSR generators have been the only source for applications that use a huge amount of random numbers. But little is known about the nature of parallelized random numbers, and unexpected behaviors in parallel Monte Carlo computations might well cause suspicion that the random numbers were not very good [3]. In such a case, it is hoped to repeat the same computation replacing the generator by another one. This explains why there exists a strong request for a parallel generator which is not based on the GFSR method. D.E. Knuth, summarizing the chapter of random numbers in his famous book [14], recommended to run each Monte Carlo program twice using different sources of random numbers.

In the previous two chapters, we have developed a method to parallelize a GFSR sequence by partitioning it into many segments of equal lengths. The point of the method is an efficient algorithm to compute terms of the sequence separated far away from known terms. For a GFSR sequence based on a primitive polynomial of degree p, the nth term is known by computation of $O(p \log n + p^2)$ steps. The extraordinary efficiency is due to the squaring property of polynomials on $GF(2)$, but this property does not help the computation of separated terms for a lagged-Fibonacci sequence using $+$, $-$ or $*$.

In this chapter, Miller-Brown's algorithm [27] is applied to parallelize lagged-Fibonacci

generators using $+$, $-$ or $*$. The algorithm was cited in the Knuth's book in the answer to an exercise (see p.637 in reference [14]), where the complexity of the algorithm was erroneously presented to be $O(p^3 \log n)$. As we will see later, the true complexity is only $O(p^2 \log n)$. This computation is quite feasible, because the value of p need not be so large for a lagged-Fibonacci generator using $+$, $-$ or $*$ as for a GFSR generator.

4.2 Lagged-Fibonacci generators

A lagged-Fibonacci generator is characterized by a recurrence of the type

$$x_k = x_{k-p} \circ x_{k-q} \tag{4.1}$$

where $p > q > 0$ and $\circ$ is some binary operation. The generator is symbolically denoted as $F(p, q, \circ)$. When $\oplus$ (bitwise exclusive-or) is used for $\circ$, it is called a GFSR generator. Other binary operations in use are $+$, $-$ and $*$ (multiplication).

We assume that $\langle x_i \rangle$ is a sequence of integers modulo 2^e. (We will take $e = 31$ in our implementation on 32bit machines.) Then the maximum possible periods for lagged-Fibonacci generators are

$$T_\oplus = 2^p - 1 \tag{4.2}$$

$$T_\pm = 2^{e-1}(2^p - 1) \tag{4.3}$$

$$T_* = 2^{e-3}(2^p - 1) \tag{4.4}$$

for $F(p, q, \oplus)$, $F(p, q, \pm)$ and $F(p, q, *)$, respectively [24]. The maximum period is achieved when some conditions are satisfied by the pair (p, q) and by the starting values for the sequence. As for the conditions on the pair (p, q), we refer the readers to reference [24]. The conditions for the starting values are

$$F(p, q, \oplus) \ : \ \text{not all zero}, \tag{4.5}$$

$$F(p, q, \pm) \ : \ \text{not all even}. \tag{4.6}$$

For $F(p, q, *)$, we will describe the conditions in §4.5. Of course, the starting values should be chosen carefully in order to get a random enough sequence.

As for randomness of these generators, we can quote Marsaglia's result of several stringent tests from reference [25]. He put lagged-Fibonacci generators $F(17, 5, \circ)$, $F(31, 13, \circ)$ and

$F(55, 24, \circ)$ to these tests. The $*$ generators passed all the tests, the $\pm$ generators gave good results except on one test, but $\oplus$ gave 'almost uniformly terrible results'. He also tested generators with long lags, say $F(607, 273, \circ)$ or $F(1279, 418, \circ)$, and obtained good results for every choice of binary operation: $+$, $-$, $*$ and even $\oplus$. The need for a large p in designing a GFSR generator is also indicated by theoretical analysis of k-distribution [6].

We may summarize the above description as follows:

1. The periods of $F(p, q, \pm)$ and $F(p, q, *)$ are much longer than that of $F(p, q, \oplus)$.

2. $F(p, q, \pm)$ and $F(p, q, *)$ are random enough for moderate values of p while much larger p is required to get a good $F(p, q, \oplus)$.

Thus it seems to be reasonable to adopt lags of moderate size, say $(p, q) = (55, 24)$, for lagged-Fibonacci generators using $+$, $-$ or $*$.

4.3 Parallelization

Random numbers used in parallel computations form a two dimensional array. The jth number generated on the ith processor (or, more generally, the ith task) will be denoted as $v_{i,j}$. An easy way to parallelize a lagged-Fibonacci generator is to use the same recurrence on each processor and generate the rows in parallel by

$$v_{i,j} = v_{i,j-p} \circ v_{i,j-q} \tag{4.7}$$

with initial values ($v_{i,j}$, $0 \leq j < p$) chosen arbitrarily for each row. However, if the initial values are given in a random manner, properties of the parallel random numbers will be unpredictable; for example, there may happen to exist pairs of rows with very large correlation.

A consistent method of parallelizing a given sequence $\langle x_k \rangle$ is to cut off segments of equal lengths from the sequence and to arrange them in rows or in columns of $v_{i,j}$ [17]. We first consider the possibility of arranging segments of length μ in columns so that $v_{i,j} = x_{i+j\mu}$. This type of parallelization can be applied to the congruential method as well as the GFSR method. If $\langle x_k \rangle$ is a congruential sequence, the sequence in rows is also a congruential sequence with a multiplier (and an adder) given by a simple computation. When the original

sequence is of GFSR type, one can choose μ to be a power of 2 so that the sequence in rows is a GFSR sequence which obeys the same recurrence as $\langle x_k \rangle$. For lagged-Fibonacci generators using $+$, $-$ or $*$, however, the recurrence for the sequence in rows is too complex to be applied for rapid generation on parallel computers.

Thus we apply the other way of arrangement; cut off segments of length ν from the original sequence and use them as rows of $v_{i,j}$ so that

$$v_{i,j} = x_{i\nu+j}. \tag{4.8}$$

In such a configuration, it is obvious that $v_{i,j}$ satisfies (4.7); each processor can generate random numbers using the same recurrence as the original sequence. If ν is large enough, there is no overlap of sequences generated on any two processors. Furthermore, we choose ν to be relatively prime to T so that the sequence appearing in the columns has also the maximum period of T.

The central problem in this method is how to give initial values for each row; each processor must prepare its initial values in a reasonable amount of time. The set of initial values for a processor is a p-tuple in the original sequence separated far away from the starting p-tuple. Efficient method is required to compute such delayed p-tuple even when the delay is very large.

4.4 Large delay for $F(p, q, \pm)$

When n is very large, it is impractical to get the p-tuple $x_n, x_{n+1}, \ldots, x_{n+p-1}$ from the initial values $x_0, x_1, \ldots, x_{p-1}$ by applying the recurrence (4.1) n times. For a linear recurrence such as $F(p, q, \pm)$, this problem can be solved using matrices. We introduce

$$X_k = \begin{pmatrix} x_k & x_{k+1} & \cdots & x_{k+p-1} \\ x_{k+1} & x_{k+2} & \cdots & x_{k+p} \\ \vdots & \vdots & & \vdots \\ x_{k+p-1} & x_{k+p} & \cdots & x_{k+2p-2} \end{pmatrix}. \tag{4.9}$$

The sequence of matrices $\langle X_k \rangle$ satisfies a recurrence relation

$$X_k = DX_{k-1}, \tag{4.10}$$

where the matrix D is independent of k and is determined solely by the recurrence for the sequence $\langle x_k \rangle$. In this matrix description, we can write

$$X_n = D^n X_0. \tag{4.11}$$

Therefore, the problem is reduced to the computation of the matrix D^n. To obtain this, let the binary representation of the delay n be

$$n = (b_{m-1} b_{m-2} \dots b_0)_2, \quad (b_{m-1} = 1). \tag{4.12}$$

Then the matrix to be computed is decomposed as

$$D^n = (\cdots ((D^{b_{m-1}})^2 D^{b_{m-2}})^2 D^{b_{m-3}} \cdots)^2 D^{b_0}. \tag{4.13}$$

Thus introducing a notation

$$n_j = (b_{m-1} b_{m-2} \dots b_{m-j})_2, \tag{4.14}$$

D^n can be obtained by repeating computation of D^β ($\beta = n_{j+1}$) from D^α ($\alpha = n_j$) for $j = 1, 2, \dots, m-1$ in order. The complexity of this algorithm is $O(p^3 \log n)$.

A more efficient method to compute a delayed sequence was proposed by J.C.P. Miller and D.J.S. Brown [27]. They introduced a sequence $\langle u_k \rangle$ defined by initial values

$$u_0 = u_1 = \cdots = u_{p-2} = 0, \quad u_{p-1} = 1 \tag{4.15}$$

and by the same linear recurrence as $\langle x_k \rangle$. From this sequence, we can construct matrices U_k in the same manner as X_k were defined. We notice that U_k involves $2p-1$ consecutive terms of $\langle u_k \rangle$ and that U_0 is a regular matrix as is seen from its triangular form. First, we write down the binary representation of the delay as in (4.12). Then we can get the p-tuple $x_n, x_{n+1}, \dots, x_{n+p-1}$ by the following algorithm.

Step 1. Obtain all the $2p-1$ terms that constitute U_1. The first $p-1$ terms $u_1, u_2, \dots, u_{p-1}$ are given in (4.15), while the other p terms $u_p, u_{p+1}, \dots, u_{2p-1}$ are determined by the recurrence.

Step 2. Compute the components of U_n. This can be done by repeating the following three substeps, which compute components of U_β ($\beta = n_{j+1}$) from those of U_α ($\alpha = n_j$), for $j = 1, 2, \dots, m-1$ in order.

2.1 (When this substep is entered, all the components of U_α are known.) Write down the p components in the zeroth (i.e. the uppermost) row of the matrix $D^\alpha = U_\alpha U_0^{-1}$.

2.2 Compute the first p terms in U_β, i.e. $u_\beta, u_{\beta+1}, \ldots, u_{\beta+p-1}$. This is done as follows. If $b_{m-j} = 0$, then obtain the zeroth row of $U_\beta = D^\alpha U_\alpha$; if $b_{m-j} \neq 0$, then after applying the recurrence to get $u_{\alpha+2p-1}$, compute the zeroth row of $U_\beta = D^\alpha U_{\alpha+1}$.

2.3 Apply the recurrence to get the remaining $p-1$ terms in U_β, i.e. $u_{\beta+p}$, $u_{\beta+p+1}$, $\ldots$, $u_{\beta+2p-2}$.

Step 3. Write down the zeroth row of the matrix $D^n = U_n U_0^{-1}$.

Step 4. Compute $x_n, x_{n+1}, \ldots, x_{n+p-1}$ as the zeroth row of the matrix $X_n = D^n X_0$.

The dominant part of this algorithm is Step 2. In particular, Step 2.2 involves calculation of the zeroth row in a product of two matrices. The complexity of this part is $O(p^2 \log n)$ which determines the complexity for the total computation.

Step 2.1 (as well as Step 3) also involves computation of the zeroth row in a product of two matrices of the form $U_k U_0^{-1}$. But for our additive or subtractive generator with recurrence

$$x_k = (x_{k-p} \pm x_{k-q}) \bmod 2^e, \tag{4.16}$$

U_0^{-1} has an especially simple structure:

$$(U_0^{-1})_{i,j} = \begin{cases} 1 & \text{if} \quad i+j = p-1 \\ \mp 1 & \text{if} \quad i+j = p-q-1 \\ 0 & \text{otherwise} \end{cases} \tag{4.17}$$

where the index convention is $0 \leq i, j < p$. This allows us to write down the required quantities as

$$(U_k U_0^{-1})_{0,i} = \begin{cases} u_{k+p-1-i} \mp u_{k+p-q-1-i} & \text{if} \quad 0 \leq i \leq p-q-1 \\ u_{k+p-1-i} & \text{if} \quad p-q \leq i \leq p-1. \end{cases} \tag{4.18}$$

Thus the complexity of Step 2.1 is only $O(p \log n)$.

In spite of the matrix notation, the algorithm can be coded using only linear arrays. Especially, an array of size p is used for the zeroth row of D^α, and an array of size $2p$ is sufficient to describe components of U_α, $U_{\alpha+1}$ and U_β.

4.5 Large delay for $F(p, q, *)$

Miller-Brown's algorithm does not apply directly to a multiplicative generator because the recurrence

$$x_k = (x_{k-p} * x_{k-q}) \bmod 2^e \tag{4.19}$$

for $F(p, q, *)$ is not linear. In fact, it would be impossible to compute remote terms in the sequence if the initial values are given arbitrarily. The parallelization, however, can be achieved by an appropriate initialization procedure based on Miller-Brown's algorithm. In order to explain this, we must review some basics of multiplicative generators [24].

If initial values $x_0, x_1, \ldots, x_{p-1}$ for $F(p, q, *)$ contain an even number, the sequence $\langle x_k \rangle$ will soon converges to zero. Thus we must choose all initial values to be odd, and then $\langle x_k \rangle$ will be a sequence of odd numbers.

In order to find the period of an $F(p, q, *)$ generator, we express the group of odd numbers (the group of residues which are relatively prime to the modulus) as a direct product of cyclic groups. For example, an odd number modulo 2^e is uniquely represented as

$$x = (-1)^y 3^z \bmod 2^e \tag{4.20}$$

$$y \in \{0, 1\}, \quad z \in \{0, 1, \ldots, 2^{e-2} - 1\}. \tag{4.21}$$

Therefore, if we write numbers in $F(p, q, *)$ as

$$x_k = (-1)^{y_k} 3^{z_k} \bmod 2^e, \tag{4.22}$$

the recurrence (4.19) is equivalent to

$$y_k = (y_{k-p} + y_{k-q}) \bmod 2 \tag{4.23}$$

$$z_k = (z_{k-p} + z_{k-q}) \bmod 2^{e-2}. \tag{4.24}$$

This explains that the maximum possible period for $\langle x_k \rangle$ is given by equation (4.4) and that the period is achieved when $\langle z_k \rangle$ has the maximum period of $2^{e-3}(2^p - 1)$.

The idea which enables parallelization of $F(p, q, *)$ is to give initial values not for $\langle x_k \rangle$ but for $\langle y_k \rangle$ and $\langle z_k \rangle$; if $y_0, y_1, \ldots, y_{p-1}$ and $z_0, z_1, \ldots, z_{p-1}$ are known, $y_n, y_{n+1}, \ldots, y_{n+p-1}$ are given by the algorithm discussed in reference [18] while $z_n, z_{n+1}, \ldots, z_{n+p-1}$ are computed by Miller-Brown's algorithm described in the previous section. In order to obtain the maximal

	$+$	$-$	$*$	$\oplus$
SUBROUTINE	INITA(I)	INITS(I)	INITM(I)	INITX(I)
FUNCTION	IRANDA()	IRANDS()	IRANDM()	IRANDX()
FUNCTION	ARANDA()	ARANDS()	ARANDM()	ARANDX()

Table 4.1: Program package of four lagged-Fibonacci generators using $+$, $-$, $*$ and $\oplus$. Each generator is provided in three subprograms, a subroutine for initialization and two functions to generate random numbers of integer and real types.

period, initial values for $\langle z_k \rangle$ must be chosen not all even. But starting values of $\langle y_k \rangle$ are completely arbitrary. In particular, we can set $y_k = 0$ for all k so that

$$x_k = 3^{z_k} \bmod 2^e. \tag{4.25}$$

A sequence of this type is convenient for our purpose of parallelization, because the initialization procedure is rather simple. Probably, properties of $\langle x_k \rangle$ as a pseudorandom sequence will not depend on the choice of $\langle y_k \rangle$. The choice of the generator for the circular group (instead of the generator 3, one can adopt any number that is congruent to 3 or 5 modulo 8) will also be irrelevant for the properties of $\langle x_k \rangle$.

4.6 Implementation

4.6.1 The program package

We have constructed a program package of lagged-Fibonacci generators on parallel computers. The programs are written in Sun FORTRAN and are intended to work on each processor, initializing and generating the sequence assigned to the processor.

The program package consists of four generators using four binary operations. It is supplied in twelve subprograms, one subroutine and two functions for each generator, as listed in Table 4.1. A generator must be initialized by calling the subroutine before generating random numbers using the functions. The integer function returns an integer in $[0, 2^{31})$, while the real function gives a floating point number normalized in $[0, 1)$.

The functions for generation have no argument, while the subroutine for initialization has one argument `I` which is used to specify the row number i of $v_{i,j}$ to be used in the processor.

For example, if each processor has an identifying number $0 \leq i_p < n_p$ on a system consisting of n_p processors, one can set `I` to $i = i_{try} * n_p + i_p$ in the i_{try}th Monte Carlo trial.

Three generators in the package are $F(55, 24, +)$, $F(55, 24, -)$ and $F(55, 24, *)$, each of which is parallelized as in equation (4.8) with

$$\nu = 2^{61} - 1. \tag{4.26}$$

Because this is a prime number, it is evident that ν is relatively prime to the periods of these generators $T_{\pm} = 2^{30}(2^{55} - 1)$ and $T_* = 2^{28}(2^{55} - 1)$. We can restrict the argument for the initialization subroutine to the range

$$0 \leq \texttt{I} < 2^{24} \quad \text{for the } \pm \text{ generators}, \tag{4.27}$$

$$0 \leq \texttt{I} < 2^{22} \quad \text{for the } * \text{ generator}, \tag{4.28}$$

so that strings of random numbers generated with different values of `I` will never overlap in a period of the original sequence.

The GFSR generator symboled by $\oplus$ in Table 4.1 is essentially the generator described in reference [18], though some changes are made to give uniform style for the four generators. It is an $F(521, 32, \oplus)$ generator parallelized with $\nu = 2^{261}$.

Each generator has its own named common block containing a circular list of p random numbers and two pointers. The contents of the common block are kept throughout the computation by `SAVE` statements specified in all the relevant programs in the package. The initialization subroutine sets the circular list to the starting values $v_{i,j}$ $(j = 0, 1, \ldots, p-1)$ for the sequence in the ith row.

All the functions for random number generation are given very similar codes. The differences are the common block assigned for each generator, the binary operation in a generation line, and whether the random integer is normalized in $[0, 1)$ or not. The prototype of the function `IRANDX` is given in reference [18].

4.6.2 Initialization for the additive generator

The subroutine `INITA` which initializes the additive generator on each processor is divided into three parts each of which is described as a subroutine:

```
SUBROUTINE INITA(I)
```

```
INTEGER IB(92)
CALL OINITA
CALL BIREPA(I,NBIT,IB)
CALL DELAYA(NBIT,IB)
END
```

First, subroutine `OINITA` fills the circular list in the common block with the initial values $x_0, x_1, \ldots, x_{p-1}$ for the original sequence. These values were set by the sequence of the most significant bit $\langle c_k \rangle$ for a congruential sequence defined by $y_{k+1} = 69069 y_k \bmod 2^{32}$ and $y_0 = 1$. Thus the binary representations are $x_0 = (c_0 c_1 \ldots c_{30})_2$, $x_1 = (c_{31} c_{32} \ldots c_{61})_2$, $\ldots$, $x_{54} = (c_{1674} c_{1675} \ldots c_{1704})_2$. `OINITA` also sets the two pointers in the common block to the appropriate initial values. (See reference [18]).

Next, subroutine `BIREPA` gives binary representation of the delay n for the segment assigned for the processor. The delay is given by the product of the row number $i =$ `I` and the delay ν between two neighboring rows in $v_{i,j}$. The value of ν specified in equation (4.26) is convenient because

$$\begin{aligned} n &= i\nu \\ &= (i-1) \cdot 2^{61} + (2^{30} - 1) \cdot 2^{31} + (2^{31} - i). \end{aligned} \tag{4.29}$$

Thus for $0 < i < 2^{31}$, the delay can be represented in 92 bits with the significant 31 bits being the representation of $i - 1$, with the middle 30 bits being all 1, and with the less significant 31 bits representing $2^{31} - i$. The bit contents are returned in the array `IB` with the number of bits (m in equation (4.12)) given in the variable `NBIT`.

Lastly, subroutine `DELAYA` replaces the p-tuple in the common block with the p-tuple delayed by n terms in the sequence. The binary representation of n is given in the arguments for the subroutine, and the algorithm explained in §4.4 is applied. In the present case, subroutine `DELAYA` replaces the values $x_0, x_1, \ldots, x_{p-1}$ in the circular list by values $x_n, x_{n+1}, \ldots, x_{n+p-1}$.

4.6.3 Initialization for the subtractive generator

The subtractive generator is initialized on each processor by subroutine `INITS` which is given as below.

```
SUBROUTINE INITS(I)
INTEGER IB(92)
CALL OINITS
CALL BIREPA(I,NBIT,IB)
CALL DELAYS(NBIT,IB)
END
```

Here `OINITS` is the same as `OINITA` except that it initializes the common block for the subtractive generator. `DELAYS` is analogous to `DELAYA` except for the common block it rewrites and for sign differences in several lines as seen in the algorithm described in §4.4.

4.6.4 Initialization for the multiplicative generator

The $*$ generator which we implemented has the simple structure given in equation (4.25). The following program initializes the multiplicative generator on each processor.

```
SUBROUTINE INITM(I)
INTEGER IB(92)
CALL OINITM
CALL BIREPA(I,NBIT,IB)
CALL DELAYM(NBIT,IB)
CALL POWERM
END
```

Subroutines `OINITM` and `DELAYM` are the same as `OINITA` and `DELAYA` respectively, except that the common block for the multiplicative generator is written or rewritten. Thus the circular list in the common block essentially contains $z_n, z_{n+1}, \ldots, z_{n+p-1}$ after the first three subroutines are executed. Finally, subroutine `POWERM` replaces each item of the circular list z_k by $x_k = 3^{z_k} \bmod 2^{31}$.

4.6.5 Timings

We measured speeds of the four generators on Fujitsu AP1000 both for initialization and for generation. Each item in Table 4.2 is the time needed to call the subprogram in the corresponding item of Table 4.1. In the estimation of these timings, each routine was called repeatedly in a `DO` loop, and the consumed time was divided by the repetition number to

	$+$	$-$	$*$	$\oplus$
Initialization	0.596 sec	0.599 sec	0.603 sec	0.544 sec
Generation (Integer)	0.882 μsec	0.882 μsec	3.34 μsec	0.841 μsec
Generation (Real)	7.21 μsec	7.21 μsec	9.67 μsec	7.17 μsec

Table 4.2: Timings, measured on Fujitsu AP1000, for a call to each subprogram listed in Table 4.1.

get the average time for a call. Initialization routines were tested with the argument `I` set to 1023. The initialization time will be slightly longer for larger values of `I`, because it is proportional to the bit number `NBIT` of the delay.

Chapter 5

Binary Method of Evaluating Remote Terms in a Recurrent Sequence

Parallelization of a random number sequence in a horizontal configuration requires an efficient algorithm of evaluating remote terms to initialize the parallel sequences. In the preceding three chapters, we have shown that efficient initialization procedures do exist for lagged-Fibonacci generators using $+$, $-$, $*$ or $\oplus$. But each of these procedures seems to be somewhat tricky or very specific to the corresponding recurrent sequence. In this chapter, we reconsider the problem of evaluating remote terms on the basis of a prescription for a general recurrent sequence.

5.1 General prescription

Let $\langle x_i \rangle$ be a sequence generated by a recurrence of pth degree from the initial values of $x_0, x_1, \ldots, x_{p-1}$. Our aim is to evaluate $x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+p-1}$ for very large n. We do this in the following three steps.

Step 1 Apply the recurrence to enumerate the values of the $p-1$ terms $x_p, x_{p+1}, \ldots, x_{2p-2}$ which succeed the initial terms.

Step 2 Compute x_n as a function of $x_0, x_1, \ldots, x_{p-1}$:

$$x_n = f_n(x_0, x_1, \ldots, x_{p-1}). \tag{5.1}$$

Step 3 Evaluate $x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+p-1}$ by substituting the values of $x_0, x_1, \ldots, x_{2p-2}$ to the right hand sides of

$$\begin{aligned} x_n &= f_n(x_0, x_1, \ldots, x_{p-1}) \\ x_{n+1} &= f_n(x_1, x_2, \ldots, x_p) \\ x_{n+2} &= f_n(x_2, x_3, \ldots, x_{p+1}) \\ &\vdots \\ x_{n+p-1} &= f_n(x_{p-1}, x_p, \ldots, x_{2p-2}). \end{aligned}$$

When we say that a function is known as in (5.1), we mean that the function has been derived using only the recurrence. Thus (5.1) is assumed to be true for arbitrary values of $x_0, x_1, \ldots, x_{p-1}$, and in fact it is identical to

$$x_{n+i} = f_n(x_i, x_{i+1}, \ldots, x_{i+p-1}) \tag{5.2}$$

where i is an arbitrary nonnegative integer. Thus the equations in the third step of the above prescription are immediate consequences of (5.1).

The problem is how to compute the function f_n in the second step. We do this by means of a binary method: Assuming that n is expressed just in m bits in binary notation, we can write

$$n = (b_{m-1} b_{m-2} \ldots b_1 b_0)_2$$

where $b_{m-1} = 1$. Then we introduce

$$n_j = (b_{m-1} b_{m-2} \ldots b_{m-j})_2$$

for $j = 1, 2, \ldots, m$, so that

$$1 = n_1 < n_2 < \cdots < n_m = n.$$

If x_{n_j} is known, we can compute $x_{n_{j+1}}$ by the prescription described below. Thus starting from $x_{n_1} = x_1$, one can obtain $x_{n_2}, x_{n_3}, \ldots, x_{n_m} = x_n$ in order, if the prescription is carried out repeatedly for $j = 1, 2, \ldots, m-1$ in this order.

For simplicity, we denote $n_j = k$. Then we are assuming that x_k is known as a function of $x_0, x_1, \ldots, x_{p-1}$:

$$x_k = f_k(x_0, x_1, \ldots, x_{p-1}).$$

If $b_{m-j-1} = 0$, then $n_{j+1} = 2k$ and we apply the above relation repeatedly to get

$$\begin{aligned} x_{2k} &= f_k(x_k, x_{k+1}, \ldots, x_{k+p-1}) \\ &= f_k(f_k(x_0, \ldots, x_{p-1}), f_k(x_1, \ldots, x_p), \ldots, f_k(x_{p-1}, \ldots, x_{2p-2})). \end{aligned}$$

The right hand side is a function of $x_0, x_1, \ldots, x_{p-1}, x_p, \ldots, x_{2p-3}, x_{2p-2}$ and we write this as

$$x_{2k} = \tilde{f}_{2k}(x_0, x_1, \ldots, x_{p-1}, x_p, \ldots, x_{2p-3}, x_{2p-2}). \tag{5.3}$$

Applying the recurrence for $x_{2p-2}, x_{2p-3}, \ldots, x_p$ in this order, we can delete the explicit dependence of the expression on these terms and arrive at the desired function

$$x_{2k} = f_{2k}(x_0, x_1, \ldots, x_{p-1}).$$

If $b_{m-j-1} = 1$ so that $n_{j+1} = 2k + 1$,

$$x_{2k+1} = \tilde{f}_{2k}(x_1, x_2, \ldots, x_{p-1}, x_p, \ldots, x_{2p-2}, x_{2p-1}).$$

We can apply the recurrence for $x_{2p-1}, x_{2p-2}, \ldots, x_p$ in this order to derive the function

$$x_{2k+1} = f_{2k+1}(x_0, x_1, \ldots, x_{p-1}).$$

Thus we can compute the function $f_{n_{j+1}}$ which describes $x_{n_{j+1}}$ in terms of $x_0, x_1, \ldots, x_{p-1}$.

5.2 Shift register generators

For a shift register sequence, we can write x_k $(k > 0)$ as a linear combination of initial terms

$$x_k = c_0 x_0 \oplus c_1 x_1 \oplus c_2 x_2 \oplus \cdots \oplus c_{p-1} x_{p-1} \tag{5.4}$$

with the coefficients c_i's being 0 or 1. This expression is trivial for $k = 1$, and the following discussion gives its proof in mathematical induction.

If we assume (5.4) is correct, then it indicates that

$$x_{2k} = c_0 x_k \oplus c_1 x_{k+1} \oplus c_2 x_{k+2} \oplus \cdots \oplus c_{p-1} x_{k+p-1}.$$

Applying (5.4) again to $x_k, x_{k+1}, \ldots, x_{k+p-1}$ in the right hand side, we can express x_{2k} in terms of $x_0, x_1, \ldots, x_{2p-2}$ as

$$\begin{aligned} x_{2k} = \quad & c_0 \left(c_0 x_0 \oplus c_1 x_1 \oplus c_2 x_2 \oplus \cdots \oplus c_{p-1} x_{p-1} \right) \\ \oplus\ & c_1 \left(c_0 x_1 \oplus c_1 x_2 \oplus c_2 x_3 \oplus \cdots \oplus c_{p-1} x_p \right) \\ \oplus\ & c_2 \left(c_0 x_2 \oplus c_1 x_3 \oplus c_2 x_4 \oplus \cdots \oplus c_{p-1} x_{p+1} \right) \\ \oplus\ & \cdots \\ \oplus\ & c_{p-1} \left(c_0 x_{p-1} \oplus c_1 x_p \oplus c_2 x_{p+1} \oplus \cdots \oplus c_{p-1} x_{2p-2} \right). \end{aligned}$$

Since $c_i c_j x_{i+j}$ and $c_j c_i x_{i+j}$ appear symmetrically for $i \neq j$,

$$\begin{aligned} x_{2k} = c_0^2 x_0 \quad &\oplus \quad 2c_0c_1x_1 \oplus (2c_0c_2 \oplus c_1^2)x_2 \\ &\oplus \quad 2(c_0c_3 \oplus c_1c_2)x_3 \oplus (2(c_0c_4 \oplus c_1c_3) \oplus c_2^2)x_4 \oplus \cdots \oplus c_{p-1}^2 x_{2p-2}. \end{aligned}$$

Now for the shift register case, this expression is simplified as

$$x_{2k} = c_0x_0 \oplus c_1x_2 \oplus c_2x_4 \oplus \cdots \oplus c_{p-1}x_{2p-2},$$

because $c_ic_i = c_i$ and $2c_ic_j = 0$ on $GF(2)$. This is the function $\tilde{f}_{2k}$ in (5.3) for a shift register random number sequence. As described in the previous section, we apply the recurrence to delete $x_{2p-2}, x_{2p-3}, \ldots, x_p$ from this expression or to delete $x_{2p-1}, x_{2p-2}, \ldots, x_p$ from

$$x_{2k+1} = c_0x_1 \oplus c_1x_3 \oplus c_2x_5 \oplus \cdots \oplus c_{p-1}x_{2p-1}.$$

Then we arrive at an expression for x_{2k} or x_{2k+1} containing only $x_0, x_1, \ldots, x_{p-1}$, which is a linear combination like the right hand side of (5.4).

Now we can write down the algorithm to evaluate a remote p-tuple in a shift register sequence. The following algorithm replaces the p-tuple in the array x_i $(0 \leq x \leq p-1)$ by the p-tuple delayed by $n = (b_{m-1}b_{m-2}\ldots b_0)_2$ terms in the sequence. The algorithm utilizes two more arrays, w_i $(0 \leq i \leq 2p-2)$ which stores the p initial values of x_i and the succeeding $p-1$ numbers, and c_i $(0 \leq i \leq 2p-1)$ for the coefficients appearing in the above discussion.

Step 1 Set $w_i \leftarrow x_i$ $(0 \leq i \leq p-1)$, then set $w_i \leftarrow x_{i-p} \oplus x_{i-q}$ $(p \leq i \leq 2p-2)$.

Step 2.1 Set $c_0 \leftarrow 0$, $c_1 \leftarrow 1$ and $c_i \leftarrow 0$ $(2 \leq i \leq p-1)$.

Step 2.2 For $j = 1, 2, \ldots, m-1$, perform the following loop:

If $b_{m-j-1} = 0$ then,

2.2.a1 for $i = p-1, p-2, \ldots, 1$, set $c_{2i} \leftarrow c_i$ and $c_{2i-1} \leftarrow 0$;

2.2.a2 for $i = 2p-2, 2p-3, \ldots, p$, set $c_{i-p} \leftarrow c_{i-p} \oplus c_i$ and $c_{i-q} \leftarrow c_{i-q} \oplus c_i$;

else

2.2.b1 for $i = p-1, p-2, \ldots, 0$, set $c_{2i+1} \leftarrow c_i$ and $c_{2i} \leftarrow 0$;

2.2.b2 for $i = 2p-1, 2p-2, \ldots, p$, set $c_{i-p} \leftarrow c_{i-p} \oplus c_i$ and $c_{i-q} \leftarrow c_{i-q} \oplus c_i$.

Step 3 For $j = 0, 1, \ldots, p-1$, set $x_j \leftarrow \Sigma_{i=0}^{p-1} c_i w_{i+j}$.

When 2.2.a1 or 2.2.b1 is entered, $x_{n_j} = x_k$ is expressed as (5.4). If $b_{m-j-1} = 0$, then 2.2.a1 is performed and after that, $x_{n_{j+1}} = x_{2k}$ is expressed as

$$x_{2k} = c_0 x_0 \oplus c_1 x_1 \oplus \cdots \oplus c_{p-1} x_{p-1} \oplus c_p x_p \oplus \cdots \oplus c_{2p-3} x_{2p-3} \oplus c_{2p-2} x_{2p-2}.$$

In 2.2.a2, we eliminate the dependence of this expression on $x_{2p-2}, x_{2p-3}, \ldots, x_p$ by applying

$$\begin{aligned} x_{2p-2} &= x_{p-2} \oplus x_{2p-q-2} \\ x_{2p-3} &= x_{p-3} \oplus x_{2p-q-3} \\ &\vdots \\ x_p &= x_0 \oplus x_{p-q} \end{aligned}$$

in this order. If $b_{m-j-1} = 1$, then 2.2.b1 and 2.2.b2 do much the same for $x_{n_{j+1}} = x_{2k+1}$. And we arrive at an expression

$$x_{n_{j+1}} = c_0 x_0 \oplus c_1 x_1 \oplus \cdots \oplus c_{p-1} x_{p-1}$$

after 2.2.a2 or 2.2.b2 is executed. The computational complexity of the second step is $O(p \log n)$, which is also the complexity for the total algorithm.

Our algorithm described above is essentially the same as that considered in chapters 2 and 3 where the algorithm was derived from a different point of view.

5.3 Additive and subtractive generators

For additive and subtractive generators, an arbitrary term is expressed as

$$x_k = (c_0 x_0 + c_1 x_1 + c_2 x_2 + \cdots + c_{p-1} x_{p-1}) \bmod 2^e \tag{5.5}$$

where the coefficients c_i's are appropriate integers. Repeated application of (5.5) leads to

$$\begin{aligned} x_{2k} &= (c_0 x_k + c_1 x_{k+1} + c_2 x_{k+2} + \cdots + c_{p-1} x_{k+p-1}) \bmod 2^e \\ &= \big(\quad c_0 \ \big(\ c_0 x_0 + c_1 x_1 + c_2 x_2 + \cdots + c_{p-1} x_{p-1} \ \big) \\ &\quad + \ c_1 \ \big(\ c_0 x_1 + c_1 x_2 + c_2 x_3 + \cdots + c_{p-1} x_p \ \big) \\ &\quad + \ c_2 \ \big(\ c_0 x_2 + c_1 x_3 + c_2 x_4 + \cdots + c_{p-1} x_{p+1} \ \big) \\ &\quad + \quad \cdots \\ &\quad + \ c_{p-1} \ \big(\ c_0 x_{p-1} + c_1 x_p + c_2 x_{p+1} + \cdots + c_{p-1} x_{2p-2} \ \big) \ \big) \bmod 2^e. \end{aligned} \tag{5.6}$$

Since $c_i c_j x_{i+j}$ and $c_j c_i x_{i+j}$ appear symmetrically for $i \neq j$,

$$x_{2k} = (c_0^2 x_0 + 2c_0 c_1 x_1 + (2c_0 c_2 + c_1^2) x_2 + 2(c_0 c_3 + c_1 c_2) x_3 + \cdots + c_{p-1}^2 x_{2p-2}) \bmod 2^e. \quad (5.7)$$

This is the function $\tilde{f}_{2k}$ in (5.3) for the additive and the subtractive sequences. We can apply the recurrence to this expression or

$$x_{2k+1} = (c_0^2 x_1 + 2c_0 c_1 x_2 + (2c_0 c_2 + c_1^2) x_3 + 2(c_0 c_3 + c_1 c_2) x_4 + \cdots + c_{p-1}^2 x_{2p-1}) \bmod 2^e. \quad (5.8)$$

to get x_{2k} or x_{2k+1} as a function of $x_0, x_1, \ldots, x_{p-1}$.

The algorithm which replaces the initial p-tuple by the delayed p-tuple can be described analogously as the algorithm given for shift register sequences in the last section. Figure 5.1 shows the algorithm for additive sequences written in FORTRAN77. We see that substeps 2.2.a1 and 2.2.b1 in the second step are more complex than the shift register case. Indeed the multiplication-addition pairs in loops 320, 322, 325, 327 are executed about $mp^2/2$ times in total, which determines the complexity of the algorithm to be $O(p^2 \log n)$.

For subtractive sequences, we can write down another subroutine `DELAYS`. We use another common block `CBRNDS` for a subtractive generator, and the addition in loop 120 and the addition in the second line of loop 331 should be replaced by subtractions.

5.4 Multiplicative generators

An arbitrary term in a multiplicative sequence is expressed as

$$x_k = (x_0^{c_0} x_1^{c_1} x_2^{c_2} \cdots x_{p-1}^{c_{p-1}}) \bmod 2^e \quad (5.9)$$

where the coefficients c_i's are appropriate nonnegative integers. Applying (5.9) repeatedly, we can derive

$$\begin{aligned} x_{2k} &= (x_k^{c_0} x_{k+1}^{c_1} x_{k+2}^{c_2} \cdots x_{k+p-1}^{c_{p-1}}) \bmod 2^e \\ &= \Big(\big(x_0^{c_0} \; x_1^{c_1} \; x_2^{c_2} \; \cdots \; x_{p-1}^{c_{p-1}} \big)^{c_0} \\ &\quad \cdot \big(x_1^{c_0} \; x_2^{c_1} \; x_3^{c_2} \; \cdots \; x_p^{c_{p-1}} \big)^{c_1} \\ &\quad \cdot \big(x_2^{c_0} \; x_3^{c_1} \; x_4^{c_2} \; \cdots \; x_{p+1}^{c_{p-1}} \big)^{c_2} \\ &\quad \cdot \quad \cdots \\ &\quad \cdot \big(x_{p-1}^{c_0} \; x_p^{c_1} \; x_{p+1}^{c_2} \; \cdots \; x_{2p-2}^{c_{p-1}} \big)^{c_{p-1}} \Big) \bmod 2^e. \end{aligned}$$

```
      SUBROUTINE DELAYA(M,IB)
C
      PARAMETER(IP=55,IQ=24)
      COMMON /CBRNDA/ JR,KR,IX(0:IP-1)
      SAVE /CBRNDA/
      INTEGER IB(0:M-1)
      INTEGER IW(0:2*IP-2),IC(0:2*IP-1)
      IF (M.EQ.0) RETURN
C STEP 1.
      DO 110 I=0,IP-1
        IW(I)=IX(I)
  110 CONTINUE
      DO 120 I=IP,2*IP-2
        IW(I)=IW(I-IP)+IW(I-IQ)
  120 CONTINUE
C STEP 2.1.
      DO 310 I=0,IP-1
        IC(I)=0
  310 CONTINUE
      IC(1)=1
C STEP 2.2.
      DO 340 J=1,M-1
        IF (IB(M-J-1).EQ.0) THEN
C 2.2.A1
            DO 321 I=2*IP-2,IP,-1
              IS=0
              DO 320 K=1,(2*IP-I-1)/2
                IS=IS+IC(IP-K)*IC(I-IP+K)
  320         CONTINUE
              IS=2*IS
              IF (IAND(I,1).EQ.0) IS=IS+IC(I/2)*IC(I/2)
              IC(I)=IS
  321       CONTINUE
            DO 323 I=IP-1,1,-1
              IS=0
              DO 322 K=0,(I-1)/2
                IS=IS+IC(K)*IC(I-K)
  322         CONTINUE
              IS=2*IS
              IF (IAND(I,1).EQ.0) IS=IS+IC(I/2)*IC(I/2)
              IC(I)=IS
  323       CONTINUE
            IC(0)=IC(0)*IC(0)
C 2.2.A2
            DO 330 I=2*IP-2,IP,-1
              IC(I-IP)=IC(I-IP)+IC(I)
              IC(I-IQ)=IC(I-IQ)+IC(I)
  330       CONTINUE
          ELSE
C 2.2.B1
            DO 326 I=2*IP-2,IP,-1
              IS=0
              DO 325 K=1,(2*IP-I-1)/2
                IS=IS+IC(IP-K)*IC(I-IP+K)
  325         CONTINUE
              IS=2*IS
              IF (IAND(I,1).EQ.0) IS=IS+IC(I/2)*IC(I/2)
              IC(I+1)=IS
  326       CONTINUE
            DO 328 I=IP-1,1,-1
              IS=0
              DO 327 K=0,(I-1)/2
                IS=IS+IC(K)*IC(I-K)
  327         CONTINUE
              IS=2*IS
              IF (IAND(I,1).EQ.0) IS=IS+IC(I/2)*IC(I/2)
              IC(I+1)=IS
  328       CONTINUE
            IC(1)=IC(0)*IC(0)
            IC(0)=0
C 2.2.B2
            DO 331 I=2*IP-1,IP,-1
              IC(I-IP)=IC(I-IP)+IC(I)
              IC(I-IQ)=IC(I-IQ)+IC(I)
  331       CONTINUE
        END IF
  340 CONTINUE
C STEP 3.
      DO 420 J=0,IP-1
        IS=0
        DO 410 I=0,IP-1
          IS=IS+IC(I)*IW(J+I)
  410   CONTINUE
        IX(J)=IAND(IS,2**31-1)
  420 CONTINUE
      END
```

Figure 5.1: Subroutine DELAYA replaces the p-tuple `IX` in the common block with the p-tuple delayed by n terms in an additive sequence. The arguments are inputs to specify the delay n, with `M` being the number of bits m and `IB` being the bit contents $b_i (0 \leq i \leq m-1)$.

Since $x_{i+j}^{c_i c_j}$ and $x_{i+j}^{c_j c_i}$ appear symmetrically for $i \neq j$,

$$x_{2k} = (x_0^{c_0^2} x_1^{2c_0c_1} x_2^{2c_0c_2+c_1^2} x_3^{2(c_0c_3+c_1c_2)} x_4^{2(c_0c_4+c_1c_3)+c_2^2} \cdots x_{2p-2}^{c_{p-1}^2}) \bmod 2^e.$$

This is the function $\tilde{f}_{2k}$ in (5.3) for the multiplicative sequence. We can apply the recurrence to this expression or

$$x_{2k+1} = (x_1^{c_0^2} x_2^{2c_0c_1} x_3^{2c_0c_2+c_1^2} x_4^{2(c_0c_3+c_1c_2)} x_5^{2(c_0c_4+c_1c_3)+c_2^2} \cdots x_{2p-1}^{c_{p-1}^2}) \bmod 2^e.$$

to get x_{2k} or x_{2k+1} as a function of $x_0, x_1, \ldots, x_{p-1}$.

From these expressions, we can conclude that the second step of the algorithm deals with the exponents of $x_0, x_1, \ldots, x_{2p-1}$ and that the manipulation is exactly the same as for the additive or the subtractive case. The subroutine `DELAYM` for a multiplicative sequence is the same as `DELAYA` in Figure 5.1 except for a few lines: The common block `CBRNDA` should be replaced by `CBRNDM`, and the line in loop 410 must be replaced by

```
IS=IS*IPOWER(IW(J+I),IC(I))
```

where `IPOWER(I,J)` is a function programmed to compute $i^j \bmod 2^{32}$ correctly in the internal 32 bit expressions. On many computer systems, `I**J` does such a computation, and we can simply replace the above expression by

```
IS=IS*IW(J+I)**IC(I)
```

on these systems.

5.5 Comparison to existing algorithms

In this chapter, a general prescription was proposed for the evaluation of remote terms in a recurrent sequence. We have applied the prescription to lagged-Fibonacci sequences. The algorithm for a shift register sequence coincides with the algorithm described in chapters 2 and 3. For additive and subtractive sequences, the algorithm given in this chapter is different from that of Miller and Brown described in chapter 4. Our algorithm for multiplicative sequences seems to be the only method that has been proposed for the evaluation of remote terms of these sequences.

Our algorithm for additive or subtractive sequence is almost twice as fast as Miller-Brown's algorithm, though computational complexity is $O(p^2 \log n)$ for either of the algorithms. For each bit positions of n, Miller-Brown's algorithm computes the uppermost row in a product of two matrices evaluating p^2 pairs of multiplication and addition. Our algorithm, however, requires only about $p^2/2$ pairs of multiplication and addition, because $c_i c_j$ and $c_j c_i$ appear symmetrically in the coefficient of x_{i+j} in (5.6). We measured the speeds of these algorithms by executing the initialization procedures repeatedly. The initialization using our algorithm was 43% faster compared to the initialization based on Miller-Brown's algorithm.

In chapter 4, we initialized the parallel multiplicative generator without evaluating remote terms in the multiplicative sequence. Our algorithm given in this chapter provides a more straightforward initialization procedure. The speed of the new initialization was faster than that of chapter 4 by 7%.

Though the prescription is quite general and applies to any recurrent sequence, it does not always serves an efficient algorithm for a computer. For example, consider a sequence defined by a quadratic recurrence

$$x_i = a x_{i-1}^2 + b x_{i-1} + c$$

and an initial value of x_0. It is easily seen that any term x_k of this sequence is given in a polynomial of x_0. By our prescription, we can derive such a polynomial for x_{2k} or x_{2k+1} from the polynomial for x_k. Evaluation of remote terms, however, is inefficient or even impossible for this case, because the degree of the polynomial increases in proportion to 2^n.

Chapter 6

Structure of Parallelized Random Number Sources

6.1 Parallelized random number source

In a horizontal configuration, properties of the original sequence is reflected in each row. And a row is continued on the next row, so that the phase difference between two neighboring rows is given by the row length. On the contrary, properties of this configuration is not evident when it is seen in the column direction. Even if the original sequence passes a series of tests for randomness, it does not always mean that randomness in columns is guaranteed. Furthermore, it would be unclear whether there is no duplication of sequences in columns. Similarly, in a vertical configuration, properties of columns as well as relations between columns are evident, but few has been known about rows. For many parallel computations, however, properties in both row and column directions are simultaneously and equally important. In fact, random numbers in a row is usually used as a sequence assigned to definite degrees of freedom, and random numbers in a column often simulate quantities of a single physical meaning.

Two-dimensional properties of a parallelized source have been studied in various aspects. One can examine correlations between every pair of rows [26]. More complete understanding will be obtained, if enough is known about the sequence of column vectors [7]. As the number of generators which work in parallel increases, however, it becomes difficult or impracticable to study parallelized sources in these aspects. In such a case, it would be natural to start examination of a parallelized source by considering its structure as a set of rows and as a set of columns. The purpose of this chapter is to provide some basic analysis on the period

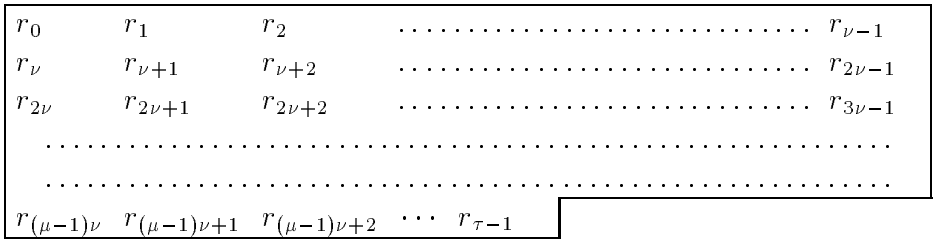


Figure 6.1: Parallelized random number source $(v_{i,j})$ which is constructed from a sequence $\langle r_i \rangle$ in a horizontal configuration.

of the sequences appearing in the orthogonal strings (columns in a horizontal configuration, rows in a vertical configuration) and on the way the parallelized source is composed of these strings.

6.2 General theory

In this section, general consideration is given on the structure of a parallelized random number source $(v_{i,j})$. Discussion in this section includes only number theoretic argument, and no assumption is made on the original generator except its period and the specification of the configuration.

All the consideration will apply both to horizontal configurations and to vertical configurations. Therefore discussion will be given only for horizontal configurations. As for vertical configurations, only main results are summarized in the final subsection.

6.2.1 Parallelized source and its extension

Assume that a sequence of random numbers $\langle r_i \rangle$ is given, and let its period be τ. (Throughout the chapter, a period is meant for the fundamental period.) As stated in the preceding section, a string of τ numbers $r_0, r_1, \ldots, r_{\tau-1}$ is called a random number source. From such a source, one can construct a parallelized random number source $(v_{i,j})$ in a horizontal configuration by partitioning it into segments of equal length ν and arranging them row by row as shown in Figure 6.1. Thus a horizontal configuration with row length ν satisfies

$$v_{i,j} = r_{i\nu+j}. \tag{6.1}$$

The number of rows in such a parallelized source is given by

$$\mu = \lceil \tau/\nu \rceil,$$

where $\lceil x \rceil$ is the least integer greater than or equal to x. Since τ is not always a multiple of ν, the last row is shorter than the other rows by

$$\kappa = \mu\nu - \tau \tag{6.2}$$

terms. These μ rows are numbered from the top as $0, 1, 2, \ldots, \mu - 1$. When the parallelized source constructed in such a way is seen as a set of columns, there are ν columns which are numbered from the left as $0, 1, \ldots, \nu - 1$. The first $\nu - \kappa$ of these columns are of length μ, whereas the remaining κ columns have length of $\mu - 1$.

Though each row involves a finite number of random numbers, the algorithm which generates these numbers defines an infinite sequence beyond the row length. The infinite sequence obtained by extending the ith row in this way is called the ith row sequence and is designated as $\langle r_j^{(i)} \rangle$. A row, which has a finite number of elements, should be strictly distinguished from a row sequence. In the same manner, each column can be extended to form an infinite sequence: If the original sequence continues to provide rows beyond its period, each column becomes an infinite sequence which will be called a column sequence. The column sequence which starts with r_j will be designated as $\langle c_i^{(j)} \rangle$ and will be called the jth column sequence. Such extension of a parallelized source $(v_{i,j})$ can be generalized to define a two-dimensional infinite sequence $\langle v_{i,j} \rangle$ with elements given for all $i \geq 0$ and $j \geq 0$ by equation (6.1). See Figure 6.2.

6.2.2 Period of a column sequence

Among various properties of a column sequence, its period can be discussed without specifying precise definition of the original generator. A quantity of essential importance is

$$g = (\nu, \tau),$$

the greatest common divisor g of ν and τ. If $\nu = g\nu'$ and $\tau = g\tau'$, then $(\nu', \tau') = 1$, that is, ν' and τ' are relatively prime. Then for any $j \geq 0$ and any $i \geq 0$,

$$c_{i+\tau'}^{(j)} = r_{(i+\tau')\nu+j} = r_{i\nu+\tau\nu'+j} = r_{i\nu+j} = c_i^{(j)},$$

so that

r_0	r_1	r_2	$\cdots$	$r_{\nu-\kappa-1}$	$r_{\nu-\kappa}$	$r_{\nu-\kappa+1}$	$\cdots$	$r_{\nu-1}$	r_ν	$\cdots$
r_ν	$r_{\nu+1}$	$r_{\nu+2}$	$\cdots$	$r_{2\nu-\kappa-1}$	$r_{2\nu-\kappa}$	$r_{2\nu-\kappa+1}$	$\cdots$	$r_{2\nu-1}$	$r_{2\nu}$	$\cdots$
$r_{2\nu}$	$r_{2\nu+1}$	$r_{2\nu+2}$	$\cdots$	$r_{3\nu-\kappa-1}$	$r_{3\nu-\kappa}$	$r_{3\nu-\kappa+1}$	$\cdots$	$r_{3\nu-1}$	$r_{3\nu}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	
$r_{(\mu-1)\nu}$	$r_{(\mu-1)\nu+1}$	$r_{(\mu-1)\nu+2}$	$\cdots$	$r_{\tau-1}$	r_0	r_1	$\cdots$	$r_{\kappa-1}$	r_κ	$\cdots$
r_κ	$r_{\kappa+1}$	$r_{\kappa+2}$	$\cdots$	$r_{\nu-1}$	r_ν	$r_{\nu+1}$	$\cdots$	$r_{\nu+\kappa-1}$	$r_{\nu+\kappa}$	$\cdots$
$r_{\nu+\kappa}$	$r_{\nu+\kappa+1}$	$r_{\nu+\kappa+2}$	$\cdots$	$r_{2\nu-1}$	$r_{2\nu}$	$r_{2\nu+1}$	$\cdots$	$r_{2\nu+\kappa-1}$	$r_{2\nu+\kappa}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	

Figure 6.2: Two-dimensional sequence $\langle v_{i,j} \rangle$ which is an extension of the parallelized source $(v_{i,j})$ in Figure 6.1.

Theorem h1 . *$\tau' = \tau/g$ is a multiple of the period of each column sequence.*

Though this theorem is rather trivial, it would be noteworthy that τ' is not always the fundamental period of $\langle c_i^{(j)} \rangle$. For example, if the original sequence is $\langle r_i \rangle = 0, 1, 0, 2, 0, 3, \ldots$ with period length 6, and if it is parallelized in a horizontal configuration with $\nu = 2$, then the period of the zeroth column sequence $\langle c_i^{(0)} \rangle = 0, 0, 0, \ldots$ is 1 and is not $\tau' = 6/2 = 3$.

According to Theorem h1, the period of a column sequence cannot exceed τ, and the period have a chance to take the maximum value τ only when $g = 1$. In fact, τ is always the fundamental period of column sequences when $g = 1$. To prove the statement, let the period of $\langle c_i^{(j)} \rangle$ be σ. Then by Theorem h1, τ is a multiple of σ. On the contrary, because $(\nu, \tau) = 1$ indicates the existence of λ which satisfies $\nu\lambda \equiv 1 \pmod{\tau}$,

$$r_{i+\sigma} = r_{(i-j+\sigma)\nu\lambda+j} = c^{(j)}_{(i-j+\sigma)\lambda} = c^{(j)}_{(i-j)\lambda} = r_{(i-j)\nu\lambda+j} = r_i,$$

so that σ is a multiple of τ. These statements are summarized as $\sigma = \tau$, and the next theorem has been proved.

Theorem h2 . *$g = 1$ is a necessary and sufficient condition for the period of each column sequence to be τ.*

Generally, a sequence of random numbers is required to have a long period. In fact, it is well known that a long period is a necessary condition for a good generator, though certainly not a sufficient one. The above two theorems indicate that the row length ν of a horizontal configuration should be chosen to be relatively prime to the period of the original sequence τ in order for the period of column sequences to be of maximal length τ.

6.2.3 Connectivity of columns

In a parallelized source of a horizontal configuration, the ith row is followed by the $(i+1 \bmod \mu)$th row. Of course, this is a loose statement of the observation that the ith row sequence is identical to the infinite sequence given by the ith row followed by the $(i+1 \bmod \mu)$th row sequence. It can be shown that there exists similar connectivity between columns. To observe this, see Figure 6.2 which shows the extended array $\langle v_{i,j} \rangle$. For example, inspection of the zeroth column sequence in this figure leads to an observation that the zeroth column $r_0, r_\nu, r_{2\nu}, \ldots, r_{(\mu-1)\nu}$ is followed by $r_\kappa, r_{\kappa+\nu}, r_{\kappa+2\nu}, \ldots$ that is the κth column. Generally,

Theorem h3 . *In a parallelized source of a horizontal configuration, the jth column is followed by the $((j+\kappa) \bmod \nu)$th column.*

According to this theorem, any column is followed by a column. Since the number of columns in a parallelized source is finite, connection of columns started from a column must return to that column after tracing some number of columns. Namely, a column sequence is embedded in the parallel source as a link of columns. If the j_1th column is followed by the j_2th column after tracing x columns $(0 \leq j_1 < \nu, 0 \leq j_2 < \nu)$,

$$j_1 + x\kappa \equiv j_2 \pmod{\nu}.$$

Since κ is expressed as in equation (6.2), $(\nu, \kappa) = (\nu, \tau) = g$. Therefore if $\kappa = \kappa' g$ and $\nu = \nu' g$, then $(\nu', \kappa') = 1$ and the above equation is equivalent to

$$x\kappa' g \equiv j_2 - j_1 \pmod{\nu' g}.$$

This equation has a solution if and only if $j_1 \equiv j_2 \pmod{g}$, and in this case x is uniquely determined modulo ν'. Furthermore, since g divides both κ and $\nu - \kappa$, every sequence involves equal number of columns with length μ and equal number of columns with length $\mu - 1$. These observations lead to the following theorem.

Theorem h4 . *The ν columns which constitute a parallelized source link to form g decoupled periodic sequences. Each of these periodic sequences is a link of $\nu' = \nu/g$ columns with ordering numbers in the same residual class modulo g. $\tau' = \tau/g$ is a multiple of the period of these sequences.*

6.2.4 Phase difference between column sequences

As was discussed in §2.2, $g = 1$ is the most interesting case. In this case, Theorem h4 indicates that all the columns of a parallelized source link to form a single periodic sequence. By Theorem h2, the period of this sequence is τ. All column sequences are equivalent to this sequence but their phase. As theorem h3 indicates, the $(i+\kappa)$th column sequence is advanced in phase by μ terms compared to the ith column sequence. A quantity of special interest would be the phase difference between two neighboring column sequences. To examine it, note that there exists λ such that $\nu\lambda \equiv 1 \pmod{\tau}$. Since $\langle c_i^{(0)} \rangle = \langle r_{i\nu} \rangle$ in a horizontal configuration and the period of $\langle c_i^{(0)} \rangle$ is τ by Theorem h2,

$$v_{i,j} = r_{i\nu+j} = r_{(i\nu+j)\nu\lambda} = c^{(0)}_{(i\nu+j)\lambda} = c^{(0)}_{i+j\lambda},$$

for any $i \geq 0, j \geq 0$. Thus the $(j+1)$th column sequence is advanced in phase by λ terms compared to the jth column sequence. All of these can be summarized in the following theorem.

Theorem h5 . *If $g = 1$, all the ν columns that constitute a parallelized source link to form a periodic sequence with period τ. In this sequence, the $(j+1)$th column sequence is advanced in phase by ν^{-1} terms compared to the jth column sequence, where ν^{-1} is the inverse element of ν modulo τ.*

Such knowledge of phase difference will be useful in the analysis of correlations between a pair of column sequences.

6.2.5 Vertical configurations

All the above consideration is applicable to vertical configurations if roles of rows and columns are exchanged. Only main conclusions are summarized in this subsection for vertical configurations.

Assume that a sequence of random numbers $\langle c_i \rangle$ is given, and let its period be τ. From a random number source $c_0, c_1, \ldots, c_{\tau-1}$, one can construct a parallelized source $(v_{i,j})$ in a vertical configuration by partitioning the source into segments of equal length μ and arranging them column by column. Thus a vertical configuration with column length μ satisfies

$$v_{i,j} = c_{i+j\mu}.$$

The number of rows in $(v_{i,j})$ is μ, whereas the number of columns is given by

$$\nu = \lceil \tau/\mu \rceil.$$

Since τ is not always divisible by μ, the last column is shorter than the other columns by

$$\kappa = \mu\nu - \tau \tag{6.3}$$

terms. If

$$g = (\mu, \tau),$$

the next two theorems apply on the period of the row sequences.

Theorem v1 . *$\tau' = \tau/g$ is a multiple of the period of each row sequence.*

Theorem v2 . *$g = 1$ is a necessary and sufficient condition for the period of each row sequence to be τ.*

As for connectivity of rows, the following theorems can be proved.

Theorem v3 . *In a parallelized source of a vertical configuration, the ith row is followed by the $((i + \kappa) \bmod \mu)$th row.*

Theorem v4 . *The μ rows which constitute a parallelized source link to form g decoupled periodic sequences. Each of these periodic sequences is a link of $\mu' = \mu/g$ rows with ordering numbers in the same residual class modulo g. $\tau' = \tau/g$ is a multiple of the period of these sequences.*

In case of $g = 1$, the following theorem gives the phase shift between two neighboring row sequences.

Theorem v5 . *If $g = 1$, all the μ rows that constitute a parallelized source link to form a periodic sequence with period τ. In this sequence, the $(i+1)$th row sequence is advanced in phase by μ^{-1} terms compared to the ith row sequence, where μ^{-1} is the inverse element of μ modulo τ.*

6.3 Applications

In this section, the general discussion given in the preceding section is applied to the congruential method and the shift register method which are the most popular generators of random numbers. Both of these parallelized methods are already in use on many vector processors as well as multiprocessors.

6.3.1 Congruential method

In the multiplicative congruential method, random numbers are defined by a recurrence

$$x_{i+1} = ax_i \bmod m.$$

Parallelization of this generator is based on the fact that its arbitrary k-wise decimation $\langle y_i \rangle = \langle x_{ki+j} \rangle$ satisfies multiplicative congruential recurrence with multiplier $b = a^k \bmod m$, that is,

$$y_{i+1} = by_i \bmod m. \tag{6.4}$$

In order to construct a parallelized source in a vertical configuration from a given congruential sequence $\langle x_i \rangle$ so that $v_{i,j} = x_{i+j\mu}$, one calculates $b = a^\mu \bmod m$ and prepares initial values $v_{i,0} = x_i$ for the rows. Then by equation (6.4), random numbers can be generated by the recurrence $v_{i,j+1} = bv_{i,j} \bmod m$ for all i in parallel, since each row is a μ-wise decimation of $\langle x_i \rangle$. On the other hand, if initial values for each row is prepared in accordance with $v_{i+1,0} = bv_{i,0} \bmod m$ where $b = a^\nu \bmod m$, the source is parallelized in a horizontal configuration so that $v_{i,j} = x_{i\nu+j}$. In this configuration, as is evident from the definition of the configuration, the recurrence which generates a row is identical to the recurrence of the original sequence $\langle x_i \rangle$, that is, $v_{i,j+1} = av_{i,j} \bmod m$.

To improve the generation rate, the modulus of the congruence m is often chosen to be a power of 2. In this case, the period of the congruential sequence is also a power of 2, which will be denoted as $\tau = 2^e$. On the other hand, the number of node processors in a multiprocessor system is often a power of 2 and the saturation vector length for a vector processor is also a power of 2. In such a situation, it is natural to parallelize a congruential sequence of period $\tau = 2^e$ in a vertical configuration with $\mu = 2^f$ $(f < e)$. But a parallelized source thus defined has $\kappa = 0$, because τ is a multiple of μ. Then by Theorem v3, each row

makes a sequence closed by itself. Though Theorem v1 states only that $\tau' = \tau/\mu = 2^{e-f}$ is a multiple of the period of a row sequence, τ' is the period itself because numbers in a period of congruential sequence are all different from each other. It is generally recognized that a congruential sequence with a short period has no good randomness. Actually, it is pointed out that a choice of such a configuration may result in misleading effects to parallel computations [26, 12].

A better design for the vertical configuration is to take $\mu = 2^f + 1$. Since μ is odd in this case, $g = (\mu, \tau) = 1$. Therefore, by Theorem v5, all the rows as a whole form a single sequence with period τ. For instance, let $\tau = 2^{30}$ and $\mu = 257 = 2^8 + 1$. In this case, there are $\nu = \lceil \tau/\mu \rceil = 4177984$ columns and the last column is shorter than the other columns by $\kappa = \mu\nu - \tau = 64$ terms. The phase difference between two neighboring row sequences is given by λ defined by $(2^8 + 1)\lambda \equiv 1 \pmod{2^{30}}$, that is, $\lambda = 2^{30} - 2^{24} + 2^{16} - 2^8 + 1 = 1057029889$. Thus the 0th row sequence is reproduced by connecting to the 0th row, the 64th row, the $2 \cdot 64$th row, the $3 \cdot 64$th row, ... , in order.

It is interesting to search configurations with $\kappa = 1$ for a congruential sequence. Since equation (6.2) or (6.3) indicates that $\kappa = 1$ is equivalent to $\tau + 1 = \mu\nu$, such a configuraion can be constructed by choosing μ or ν to be a factor of $\tau + 1$. For example, consider again the case of $\tau = 2^{30}$. Since $\tau + 1 = 5^2 \cdot 13 \cdot 41 \cdot 61 \cdot 1321$, one can choose $\mu = 5 \cdot 61 = 305$, so that $\nu = 5 \cdot 13 \cdot 41 \cdot 1321 = 1047553$ and $\kappa = 1$. Configurations with $\kappa = 1$ have especially simple structure, as will be discussed in the following subsection.

6.3.2 Shift register method

In the shift register method, a sequence of random numbers is generated in terms of a recurrence

$$x_i = x_{i-p} \oplus x_{i-q}, \tag{6.5}$$

where $\oplus$ designates bitwise exclusive-or operation. The period of this sequence $\langle x_i \rangle$ is $\tau = 2^p - 1$ if and only if the characteristic polynomial $f(x) = x^p + x^q + 1$ is primitive. The k-wise decimation $\langle y_i \rangle = \langle x_{ki+j} \rangle$ of $\langle x_i \rangle$ has the maximal period τ if and only if k is relatively prime to τ, though its characteristic polynomial does not always agree with $f(x)$. That k is a power of 2 is a necessary and sufficient condition for these two polynomials to agree [8].

When $\langle x_i \rangle$ given in equation (6.5) is used to construct a vertical configuration so that

$v_{i,j} = x_{i+j\mu}$, it is usual to take μ to be a power of 2, because only in this case each row of $(v_{i,j})$ is generated by the recurrence

$$v_{i,j} = v_{i,j-p} \oplus v_{i,j-q} \tag{6.6}$$

which includes only one exclusive-or. As for a horizontal configuration $v_{i,j} = x_{i\nu+j}$, the recurrence for each row is equation (6.6) for any value of ν. However, it is convenient to take ν to be a power of 2, because the computation of initial values for each row is simplest for such a value of ν [17].

Since the period of a shift register generator is $\tau = 2^p - 1$, the condition $g = 1$ is clearly satisfied by these configurations. Moreover, if the column length μ of a vertical configuration is a power of 2, the row length ν is also a power of 2. Similarly, if the row length ν of a horizontal configuration is a power of 2, the column length μ is also a power of 2. In either configuration, the parallelized source has $\kappa = 1$ which means that its form is a rectangle with the single number at the lower right removed.

A parallelized source in this form is of particular interest because its structure is especially simple. For example, consider a horizontal configuration constructed from $\langle r_i \rangle = \langle x_i \rangle$. Since $\kappa = 1$, Theorem h3 states that each column is followed by the column just neighboring on the right. By linking all the columns from the 0th column to the $(\nu - 1)$th column in order, one gets a period for the 0th column sequence $\langle c_i^{(0)} \rangle$ in accordance with Theorem h2 and Theorem h5. Thus this configuration is considered as a vertical configuration constructed from $\langle c_i \rangle = \langle c_i^{(0)} \rangle$. Note that $\langle c_i \rangle$ is a shift register random numbers that obeys the same recurrence as the original sequence $\langle r_i \rangle$. If a parallelized source is constructed in this form, there is a chance to give enough randomness to both of $\langle r_i \rangle$ and $\langle c_i \rangle$. For example, it is possible to give the maximal k-distribution property [6] for the two sequences simultaneously. Furthermore, various standard empirical tests can be performed to both of these sequences to check their randomness.

Chapter 7

Parallelized Feedback Shift Register Generators

Equipartitioning of a GFSR sequence has been discussed by some authors independently [17, 18, 28, 1]. There are four types of equipartitioned GFSR generators, because the GFSR sequence can be of Tausworthe type or of Lewis-Payne type and the configuration can be horizontal or vertical. Furthermore, there is another method proposed by Fushimi [5, 7] which is also based on a feedback shift register sequence. In this chapter, all of these generators are discussed within the framework of the parallelized feedback shift register (PFSR) generators. In terms of the PFSR description, a parallel generator of an arbitrary type can be interpreted as a generator of any other type. As an application of such reinterpretation, we analyze the bit structure of the PFSR generator which is intimately related to the correlation functions of the random numbers.

7.1 FSR sequences

Assume that

$$f(x) = 1 + c_1 x + c_2 x^2 + \cdots + c_p x^p \tag{7.1}$$

is a primitive pth degree polynomial over $GF(2)$. Then a sequence of bits $\langle a_n \rangle$ defined by a linear recurrence

$$a_n = (c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_p a_{n-p}) \bmod 2 \tag{7.2}$$

with initial values $(a_0, a_1, \ldots, a_{p-1})$ not all zero is called a feedback shift register (FSR) sequence generated by $f(x)$. Most often a FSR sequence generated by a primitive trinomial

$$f(x) = 1 + x^q + x^p \qquad (0 < q < p) \tag{7.3}$$

is used because of generation efficiency.

The following properties of FSR sequences are relevant to the following discussion. The proofs of these properties and more detailed description of FSR sequences are found in reference [9].

1. Period: The period of $\langle a_n \rangle$ is

$$T = 2^p - 1. \tag{7.4}$$

   In a period, the p-tuple of p consecutive elements $(a_n, a_{n+1}, \ldots, a_{n+p-1})$ takes all patterns except $(0, 0, \ldots, 0)$. Thus a period is composed of $2^{p-1} - 1$ zeros and 2^{p-1} ones.

2. Cycle-and-add property: The sequence defined by adding some phase shifts of $\langle a_n \rangle$ modulo 2 is equivalent to a phase shift of $\langle a_n \rangle$, provided it is not identical to a sequence of zeros.

3. Autocorrelation: Neglecting terms of $O(1/T)$ and $O(1/T^2)$, the average and the autocorrelation function of $\langle a_n \rangle$ are given by

$$\bar{a} = \frac{1}{T} \sum_{n=0}^{T-1} a_n \approx \frac{1}{2} \tag{7.5}$$

   and

$$R_a(s) = \frac{1}{T} \sum_{n=0}^{T-1} (a_n - \bar{a})(a_{n+s} - \bar{a}) \approx \begin{cases} \frac{1}{4} & \text{if } s \equiv 0 \pmod{T} \\ 0 & \text{if } s \not\equiv 0 \pmod{T}. \end{cases} \tag{7.6}$$

   These are just the average and the autocorrelation expected from a sequence of truly random bits.

4. Decimation: Let G be the Abelian group of residues which are relatively prime to the modulus T. The number of elements in G is known as Euler's totient function $\phi(T)$. Because T is given by (7.4), the set $C_0 = \{1, 2, 4, \ldots, 2^{p-1}\}$ is a subgroup of G. Let the $r = \phi(T)/p$ cosets of C_0 be $C_0, C_1, \ldots, C_{r-1}$. The k-wise decimation $\langle a_n^{(k)} \rangle = a_0, a_k, a_{2k}, \ldots$ is a FSR sequence with the same period as $\langle a_n \rangle$, if and only if k

is relatively prime to T. Two FSR sequences $\langle a_n^{(k)} \rangle$ and $\langle a_n^{(k')} \rangle$ are phase shifts of each other, if and only if $k \bmod T$ and $k' \bmod T$ belong to the same coset. There exist r distinct primitive polynomials of degree p, and they are in one to one correspondence to the r cosets. Let the polynomial corresponding to C_i be f_i. Then $\langle a_n^{(k)} \rangle$ is generated by f_i if $k \bmod T \in C_i$.

7.2 GFSR generators

A generalized feedback shift register (GFSR) sequence is a pseudorandom sequence which is based on a FSR sequence. There are two types of GFSR sequences, the Tausworthe type [29] and the Lewis-Payne type [15], and the latter type is usually called the GFSR sequences. In this section, however, we adopt a more general definition of GFSR sequences and show that the two types are essentially equivalent. The general treatment of GFSR generators as given here was once discussed by Fushimi [5]. It will be extended to the parallel cases to define PFSR generators in the next section.

In a random sequence of l-bit fixed-point numbers, let the kth bit of the ith number be b_{ik} $(0 \le i, 0 \le k < l)$. We define a GFSR sequence as a sequence whose bit contents are given in terms of a FSR sequence $\langle a_n^{(u)} \rangle$ as

$$b_{ik} = a_{ix+kw}^{(u)}. \tag{7.7}$$

Here u, x and w are all assumed to be relatively prime to T. We will designate the GFSR sequence in (7.7) by $G(u; x, w)$. When v is relatively prime to T,

$$a_{ix+kw}^{(u)} = a_{iux+kuw} = a_{iuvv^{-1}x+kuvv^{-1}w} = a_{iv^{-1}x+kv^{-1}w}^{(uv)} \tag{7.8}$$

so that we have a formula

$$G(u; x, w) = G(uv; v^{-1}x, v^{-1}w). \tag{7.9}$$

Here v^{-1} denotes the inverse element of $v \bmod T$ in the group G.

Both the Tausworthe type and the Lewis-Payne type are special cases of our general GFSR sequences. In fact, the Tausworthe sequence constructed from $\langle a_n^{(u)} \rangle$ with word-to-word phase difference σ is defined as $b_{ik} = a_{i\sigma+k}^{(u)}$, so that

$$T(u; \sigma) = G(u; \sigma, 1). \tag{7.10}$$

And the Lewis-Payne sequence with phase difference between two neighboring bit positions τ is defined by $b_{ik} = a^{(u)}_{i+k\tau}$, so that

$$L(u;\tau) = G(u;1,\tau). \tag{7.11}$$

Here σ and τ are assumed to be relatively prime to T.

Conversely, any GFSR sequence $G(u;x,w)$ can be interpreted as a Tausworthe sequence and as a Lewis-Payne sequence. To see this, we use (7.9) to get

$$\begin{aligned} G(u;x,w) &= G(uw;w^{-1}x,1) &(7.12)\\ &= G(ux;1,x^{-1}w). &(7.13)\end{aligned}$$

Applying (7.10) and (7.11) to the right-hand sides, we arrive at

$$\begin{aligned} G(u;x,w) &= T(uw;w^{-1}x) &(7.14)\\ &= L(ux;x^{-1}w). &(7.15)\end{aligned}$$

From the above discussion, we can conclude that T, L and G are all equivalent to each other in the sense that a generator of one type can be identified as a generator of any other type. T is translated into L by equations (7.10) and (7.15) as

$$T(u;\sigma) = L(u\sigma;\sigma^{-1}). \tag{7.16}$$

Similarly, the translation of L into T is given by equations (7.11) and (7.14) as

$$L(u;\tau) = T(u\tau;\tau^{-1}). \tag{7.17}$$

7.3 PFSR generators

As discussed previously, random numbers in a parallel environment form a two dimensional array. Let the kth bit of the random number v_{ij} be b_{ijk} $(0 \le i,\ 0 \le j,\ 0 \le k < l)$. Assuming that u, x, y, w are relatively prime to T, we construct an array of random numbers from a FSR sequence $\langle a^{(u)}_n \rangle$ as

$$b_{ijk} = a^{(u)}_{jx+iy+kw}. \tag{7.18}$$

This generator will be denoted by $P(u;x,y,w)$. In this generator, the bit sequence generated in every bit position in every row is a phase shift of a single FSR sequence so that these bit

sequences can be generated by a single recurrence in parallel. Therefore, a parallel generator of this type will be called a parallelized feedback shift register (PFSR) generator. In the same way as we derived (7.9), we see that

$$P(u;x,y,w) = P(uv;v^{-1}x, v^{-1}y, v^{-1}w), \tag{7.19}$$

where v is assumed to be relatively prime to T.

Generally in the equipartition method, we cut off segments of equal length from a single pseudorandom sequence $X = \langle x_n \rangle$ and arrange them in rows or in columns to construct a two dimensional array. If the original sequence is arranged in the row direction as $v_{ij} = x_{i\nu+j}$, we say the original sequence is parallelized in a horizontal configuration with row length ν and designate the array by $H(X;\nu)$. If the segments are arranged in columns so that $v_{ij} = x_{i+j\mu}$, we call the parallelization a vertical configuration with column length μ and designate the generator by $V(X;\mu)$. In applying the equipartition method to GFSR sequences, we have four possibilities because the original sequence can be either of the Tausworthe type (T) or of the Lewis-Payne type (L) and the configuration can be horizontal (H) or vertical (V). All of these possibilities are easily seen to be special cases of the PFSR generators:

$$H(T(u;\sigma),\nu) = P(u;\sigma,\sigma\nu,1) \tag{7.20}$$

$$V(T(u;\sigma),\mu) = P(u;\sigma\mu,\sigma,1) \tag{7.21}$$

$$H(L(u;\tau),\nu) = P(u;1,\nu,\tau) \tag{7.22}$$

$$V(L(u;\tau),\mu) = P(u;\mu,1,\tau). \tag{7.23}$$

For example, the first equation (7.20) is derived from the observation that $H(T(u;\sigma),\nu)$ arranges a Tausworthe sequence ($w = 1$) in the row direction ($x = \sigma$) with a phase difference ν between two neighboring rows (so that $y = x\nu = \sigma\nu$).

Next we show that the PFSR generator $P(u;x,y,w)$ can be interpreted as a special case of any of the four equipartition method. By (7.19),

$$P(u;x,y,w) = P(uw;w^{-1}x, w^{-1}x\cdot x^{-1}y, 1) \tag{7.24}$$

$$= P(uw;w^{-1}y\cdot y^{-1}x, w^{-1}y, 1) \tag{7.25}$$

$$= P(ux;1, x^{-1}y, x^{-1}w) \tag{7.26}$$

$$= P(uy;y^{-1}x, 1, y^{-1}w). \tag{7.27}$$

Comparing the right-hand sides with (7.20), (7.21), (7.22) and (7.23), we can read these equations as

$$\begin{aligned} P(u;x,y,w) &= H(T(uw;w^{-1}x),x^{-1}y) && (7.28)\\ &= V(T(uw;w^{-1}y),y^{-1}x) && (7.29)\\ &= H(L(ux;x^{-1}w),x^{-1}y) && (7.30)\\ &= V(L(uy;y^{-1}w),y^{-1}x). && (7.31) \end{aligned}$$

Thus the four equipartition method and the PFSR method are equivalent to each other in the sense that any one of them are reinterpreted as a special case of any other one.

Fushimi [5, 7] proposed a method of parallel random number generation based on a FSR sequence which have no correlation between parallelized sequences. Consider a multiprocessor system composed of n_p processors. Fushimi's method is to consider a Tausworthe sequence $T(u;\sigma)$ with the word-to-word phase shift σ ($\sigma \geq n_p l$), then to slice it into n_p sequences of l-bit word length, and finally to distribute them to the n_p processors. The merit of this method is that correlation functions between any two row sequences vanish up to a huge lag provided $\sigma \sim n_p l$. Now, it is easy to see that

$$F(u;\sigma,l) = P(u;\sigma,l,1). \tag{7.32}$$

Thus Fushimi's method is also a special case of the PFSR method. The inverse translation, though rather formal, is given by

$$P(u;x,y,z) = F(uz;z^{-1}x,z^{-1}y). \tag{7.33}$$

7.4 Phase shift analysis of the PFSR generators

All the parallel bit sequences appearing in the row direction or in the column direction of a PFSR array are phase shifted versions of a single FSR sequence. In applying a PFSR generator, one must take care not to use a FSR sequence duplicately. The aim of this section is to derive conditions for the absence of such duplication. Before starting the discussion, we must fix the range of the random number array to be used in a parallel computation. We assume that consecutive n_r rows are generated in parallel and that n_c random numbers are used in each row.

By (7.30), the PFSR array $P(u;x,y,z)$ is a horizontal configuration of a Lewis-Payne sequence with row length $x^{-1}y$, and the phase difference between two neighboring bit positions of the Lewis-Payne sequence is given by $x^{-1}w$. Therefore, the two bit sequences appearing in the k_1th bit position of the i_1th row and in the k_2th bit position of the i_2th row are phase shifts of $\langle a_n^{(ux)} \rangle$ with the phase difference

$$s_r(i,k) = (i \cdot x^{-1}y + k \cdot x^{-1}w) \bmod T, \tag{7.34}$$

where $i = i_2 - i_1$ and $k = k_2 - k_1$. Let Δ_r be the minimum of these phase differences taken over all i and k such that

$$|i| < n_r, \quad |k| < l, \quad (i,k) \neq (0,0). \tag{7.35}$$

We require that there is no duplication of bit sequence in the row direction in the used region of the random number array. This condition is expressed as

$$\Delta_r \geq n_c. \tag{7.36}$$

In the same way, any duplication of bit sequence should be avoided in the column direction. As is seen from (7.31), $P(u;x,y,z)$ is a vertical configuration of a Lewis-Payne sequence with column length $y^{-1}x$, where the Lewis-Payne sequence is constructed with the phase difference between neighboring bit positions of $y^{-1}w$. Thus the phase difference between the two bit sequences appearing in the k_1th bit position of the j_1th column and in the k_2th bit position of the j_2th column is given by

$$s_c(j,k) = (j \cdot y^{-1}x + k \cdot y^{-1}w) \bmod T, \tag{7.37}$$

where $j = j_2 - j_1$ and $k = k_2 - k_1$. Let Δ_c be the minimum of these phase differences taken over all j and k such that

$$|j| < n_c, \quad |k| < l, \quad (j,k) \neq (0,0). \tag{7.38}$$

Then a PFSR generator should satisfy the condition

$$\Delta_c \geq n_r. \tag{7.39}$$

(7.36) and (7.39) are just the conditions for the PFSR generator to have good correlation properties in the row direction and in the column direction. For example, the correlation function with lag s between the i_1th and the i_2th row sequences is expressed as

$$R_{i_1 i_2}(s) = \sum_{k_1=0}^{l-1} \sum_{k_2=0}^{l-1} 2^{-(k_1+k_2+2)} R_a(s + s_r(i_2 - i_1, k_2 - k_1)). \tag{7.40}$$

Because the autocorrelation function R_a of the FSR sequence $\langle a_n \rangle$ is two-valued as in (7.6), the correlation function for $|s| < \Delta_r$ is given by

$$R_{i_1 i_2}(s) \approx \begin{cases} \frac{1}{12}(1 - 2^{-2l}) & \text{if} \quad i_1 = i_2 \quad \text{and} \quad s \equiv 0 \pmod{T} \\ 0 & \text{otherwise,} \end{cases} \tag{7.41}$$

while the correlation function for $|s| = \Delta_r$ differs from this expression for at least one pair of i_1 and i_2. Because (7.41) is the expected correlation function for truly random sequences of l-bit numbers, we can interpret (7.36) as the condition for the PFSR generator to have good correlation properties.

7.5 Examples

In this section, we apply the analysis given in the preceding section to four examples of parallel generators which have appeared in the literature. It will be shown that two of them violate the conditions so that they have difficulty in correlation properties.

In the condition (7.36), the amount of random numbers n_c used in a row sequence varies from application to application. For the examples we are examining, however, it suffices to notice that n_c is much larger than 2^{10} and never exceeds 2^{60}. Except for the generator in Example 2, n_r in the condition (7.39) is assumed to be equal to the number of processors n_p.

Example 1. Fushimi's method [5, 7] was proposed in the purpose of constructing multiple sequences whose correlation functions vanish up to the largest possible lags. He did not give any complete set of parameters to specify a parallel generator. We assume here that the FSR sequence is generated by a primitive trinomial with $(p, q) = (521, 32)$, $l = 32 = 2^5$, $n_r = n_p = 1024 = 2^{10}$ and $\sigma = n_p l = 2^{15}$. Thus our example is $F(u; 2^{15}, 2^5)$. Applying (7.32), we know that this is a PFSR generator with

$$x = 2^{15}, y = 2^5, w = 1. \tag{7.42}$$

Then by (7.34) and (7.35)

$$s_r(i,k) = (i \cdot 2^{511} + k \cdot 2^{506}) \bmod (2^{521} - 1), \quad |i| < 2^{10}, |k| < 2^5, (i,k) \neq (0,0), \tag{7.43}$$

and by (7.37) and (7.38)

$$s_c(j,k) = (j \cdot 2^{10} + k \cdot 2^{516}) \bmod (2^{521} - 1), \quad |j| < n_c, |k| < 2^5, (j,k) \neq (0,0). \tag{7.44}$$

The minima of these phase differences are

$$\Delta_r = s_r(-2^{10} + 1, -2^5 + 1) = 2^{506} - 1 \tag{7.45}$$

and

$$\Delta_c = s_c(1,0) = 2^{10} = n_r. \tag{7.46}$$

These equations show that this generator satisfies the two conditions (7.36) and (7.39). In particular, the minimum (7.45) is the largest possible value of Δ_r for the $1024 \times 32 = 2^{15}$ bit sequences appearing in the row direction.

The k-distributivity of a row sequence is not guaranteed even for $k = 1$ because $\lfloor p/\sigma \rfloor = 0$. It should be checked whether the row sequences are k-distributed up to $k = \lfloor p/l \rfloor$ [6].

Example 2. In reference [18], a sample generator for multiprocessor systems is given in FORTRAN. The generator equipartitions a Tausworthe sequence with $\sigma = 32$ and $l = 31$ into a horizontal configuration with $\nu = 2^{261}$. The Tausworthe sequence is based on a FSR sequence generated by a primitive trinomial $(p,q) = (521, 32)$. One of the merits of this generator is that the array has a huge number of rows which can be used in parallel. In the FORTRAN program given in the reference, each processor can specify any of the first $n_r = 2^{31}$ rows of the array for its use. Thus it is easy to repeat many independent Monte Carlo experiments using separate parts of the array. Another merit of the generator is that the maximal k-distributivity of the row sequences is evident from its construction. The PFSR parameters for this generator $H(T(u;2^5), 2^{261})$ is given by (7.20) as

$$x = 2^5, y = 2^{266}, w = 1. \tag{7.47}$$

Therefore

$$s_r(i,k) = (i \cdot 2^{261} + k \cdot 2^{516}) \bmod (2^{521} - 1), \quad |i| < 2^{31}, |k| < 31, (i,k) \neq (0,0), \tag{7.48}$$

and

$$s_c(j,k) = (j \cdot 2^{260} + k \cdot 2^{255}) \bmod (2^{521} - 1), \quad |j| < n_c, |k| < 31, (j,k) \neq (0,0). \tag{7.49}$$

The minimum values are

$$\Delta_r = s_r(1,0) = 2^{261} \tag{7.50}$$

and

$$\Delta_c = s_c(0,1) = 2^{255}. \tag{7.51}$$

Both of these are extraordinarily large and the two conditions are safely satisfied.

Example 3. A VLSI implementation of a parallel random number generator with $l = 16$ was introduced in reference [28]. It equipartitions a Tausworthe sequence with $p = 127$ and $\sigma = 16$ into n_p segments to make a horizontal configuration with $\nu \simeq 2^{127}/n_p$. The parallel system involves three Monte Carlo computers each consisting of eight microprocessors. Because each Monte Carlo computer can perform its own task, the number of processors which generate random numbers in parallel is $n_p = 8$, 16 or 24.

Let $n_p = 8$. Then the generator is $H(T(u;2^4), 2^{124})$ so that the PFSR parameters are given from (7.20) as

$$x = 2^4, y = 2, w = 1. \tag{7.52}$$

Substituting these values into (7.34) and (7.35), we get

$$s_r(i,k) = (i \cdot 2^{-3} + k \cdot 2^{-4}) \bmod (2^{127} - 1), \quad |i| < 2^3, |k| < 2^4, (i,k) \neq (0,0). \tag{7.53}$$

The minimum of these values is

$$\Delta_r = s_r(i, -2i) = 0 \quad (0 < |i| < 2^3). \tag{7.54}$$

For example, the bit sequence appearing in the second bit position of the ith row is identical to the bit sequence appearing in the zeroth bit position of the $(i+1)$th row. Thus there is a large correlation between any two neighboring rows. By (7.37) and (7.38),

$$s_c(j,k) = (j \cdot 2^3 + k \cdot 2^{-1}) \bmod (2^{127} - 1), \quad |j| < n_c, |k| < 2^4, (j,k) \neq (0,0). \tag{7.55}$$

The minimum of these is

$$\Delta_c = s_c(0,2) = 1. \tag{7.56}$$

For example, the phase difference between two bit sequences in the zeroth and the second bit positions in each column sequence is only one.

We can repeat similar analysis assuming $n_p = 16$ or 24. The result is much the same as in the case of $n_p = 8$. In fact, row sequences assigned to eight of the processors are (almost) the same for the three cases, because both 16 and 24 are multiples of 8.

As the authors of reference [28] checked by a series of statistical tests, each row sequence seems to have no problem in randomness. But there exist strong correlations between some pairs of row sequences, and the column sequences have large autocorrelations.

Example 4. In reference [1], a Lewis-Payne sequence with $(p, q) = (607, 334)$ is parallelized in a vertical configuration with $\mu = n_r = n_p = 8192 = 2^{13}$. The phase difference between two neighboring bit positions of the Lewis-Payne sequence was chosen to be $pn_p = 2^{13}p$ for the sake of initialization efficiency. Since this generator is $V(L(u; 2^{13}p), 2^{13})$, the PFSR parameters are given by (7.23) as

$$x = 2^{13}, y = 1, w = 2^{13}p. \tag{7.57}$$

Therefore

$$s_r(i,k) = (i \cdot 2^{594} + k \cdot p) \bmod (2^{607} - 1), \quad |i| < 2^{13}, |k| < l, (i,k) \neq (0,0), \tag{7.58}$$

and

$$s_c(j,k) = (j \cdot 2^{13} + k \cdot 2^{13}p) \bmod (2^{607} - 1), \quad |j| < n_c, |k| < l, (j,k) \neq (0,0). \tag{7.59}$$

The minima are

$$\Delta_r = s_r(0,1) = p \tag{7.60}$$

and

$$\Delta_c = s_c(-kp, k) = 0 \quad (0 < |k| < l). \tag{7.61}$$

Thus neither of the conditions (7.36) and (7.39) is satisfied.

Lewis and Payne [15] argued that the phase difference between two neighboring bit positions τ should be $100p$ or more. But the discussion in the preceding section, when applied to the Lewis-Payne sequences, leads to a stronger requirement

$$\tau \geq n \tag{7.62}$$

where n is the number of random numbers to be generated.

To illustrate the importance of the phase difference, we consider the simulation of isotropic random walk on a square lattice. In each step of the simulation, one generates a random number and moves the point to the nearest neighboring site on the right, above, on the left or below according as the random number falls in $[0, \frac{1}{4})$, $[\frac{1}{4}, \frac{1}{2})$, $[\frac{1}{2}, \frac{3}{4})$ or $[\frac{3}{4}, 1)$. We mark the final position (x, y) of a point which starts from the origin and suffers $n_{step} = 5000$ steps of random walk.

Each part of Figure 7.1 shows the results of consecutive simulations of 100 points using a row sequence of the PFSR generator in Example 2 (Part (a)) or in Example 4 (Part (b)). If the row sequence is random enough, points will distribute isotropically around the origin. Part (a) of the figure shows no singular behavior, but part (b) is obviously anomalous.

We can explain the anomaly caused by a row sequence of Example 4 as follows: The random walk described above is determined by the most significant two bits of the random numbers. In a PFSR generator, the two bits in the ith random number can be written as a_i and $a_{i+\tau}$ where $\langle a_n \rangle$ is a FSR sequence of pth degree. If we write $\alpha_i = (-1)^{a_i}$, $x+y$ increases by one when $\alpha_i = 1$ and decreases by one when $\alpha_i = -1$ so that

$$x + y = \sum_{i=1}^{n_{step}} \alpha_i. \tag{7.63}$$

And $x - y$ increases by one when α_i and $\alpha_{i+\tau}$ are of the same sign while it decreases by one when they are of the opposite signs. From this, we conclude

$$x - y = \sum_{i=1}^{n_{step}} \alpha_i \alpha_{i+\tau}. \tag{7.64}$$

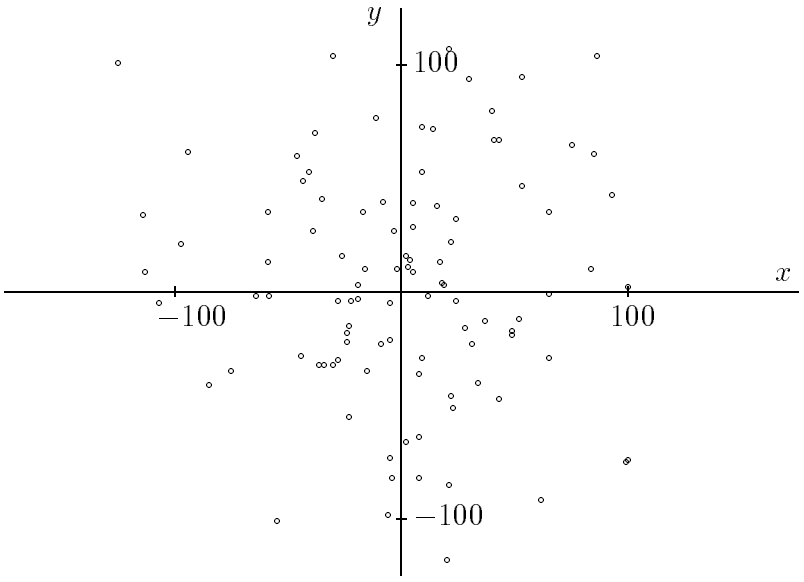
By the cycle-and-add property, we can write

$$x - y = \sum_{i=1}^{n_{step}} \alpha_{i+r} \tag{7.65}$$

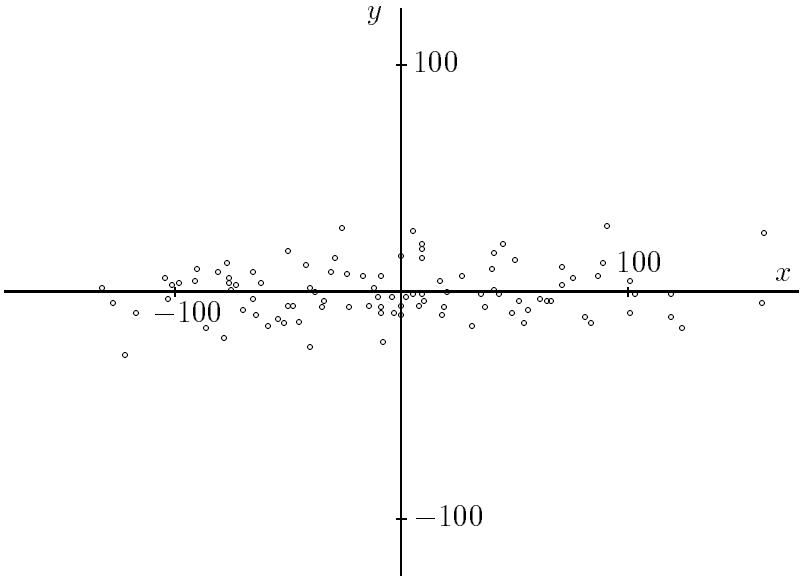
with an appropriate constant r. Therefore, by (7.63) and (7.65),

$$x = \frac{1}{2}\left(\sum_{i=1}^{n_{step}} \alpha_i + \sum_{i=1}^{n_{step}} \alpha_{i+r} \right), \tag{7.66}$$

$$y = \frac{1}{2}\left(\sum_{i=1}^{n_{step}} \alpha_i - \sum_{i=1}^{n_{step}} \alpha_{i+r} \right). \tag{7.67}$$



(a)



(b)

Figure 7.1: Two simulations of isotropic random walk on a square lattice. Each small circle indicates the final position of a point which started from the origin and experienced 5000 steps of random walk. Such a simulation was repeated for 100 points using (a) a sequence in Example 2, and (b) a sequence in Example 4.

In a row sequence of Example 4, where $\tau = p$ as shown in (7.60), the recurrence equation indicates $r = p - q$ so that the two summations in the parentheses overlap largely when $n_{step} \gg p - q$. This shows why y values in Figure 7.1(b) are relatively small for most points.

The anomalous behavior of the simulation will be observed for arbitrary τ's of the form $2^e p$ provided $n_{step} \gg 2^e(p-q)$. In fact, we can show $r = 2^e(p-q)$ in these cases by applying $a_n = (a_{n-2^e p} + a_{n-2^e q}) \bmod 2$ to the right-hand side of (7.64). Thus τ of order $100p$ is not always sufficient to get a good Lewis-Payne sequence. Rather we should require (7.62) which is the analogue of (7.36).

Chapter 8

Conclusion

Monte Carlo simulations have always been one of the main tasks for the most advanced computers of the age. In these years, the advent of highly parallel machines with supreme performances urges efforts to develop various techniques of handling Monte Carlo simulations in parallel environments. Especially, the problem of parallel random number generation is of fundamental importance, because random numbers form the basis for all Monte Carlo simulations.

In this work, we discussed the equipartition method for generating random numbers on parallel computers. We parallelized lagged-Fibonacci generators with $\oplus$, $+$, $-$ or $*$ in horizontal configurations. The shift register generators were shown to be parallelizable in vertical configurations as well. Algorithms for evaluating remote terms in lagged-Fibonacci sequences were given to initialize the table of random numbers. The complexity of these algorithms were $O(p \log n)$ for shift register sequences and $O(p^2 \log n)$ for other kinds of lagged-Fibonacci sequences. The initialization procedures based on these algorithms are very efficient, and our parallel generators can be served for general use.

One of the merits in the equipartition method is that it introduces a definite structure in the array of random numbers. This is very important because the structure permits theoretical analysis from which we can draw useful information in designing a good parallelization. We performed such analysis for a general parallelized source assuming only that the original sequence is periodic. It was shown that a parallelized source in a horizontal configuration can also be a parallelized source of another periodic sequence in a vertical configuration. The condition for such a configuration is that the length of the equipartitioned strings is prime to the period of the original sequence.

For shift register generators, structure of the parallelized source was investigated in more detail. We introduced the PFSR formalism to describe various parallelized shift register random numbers from a unified point of view. In parallelizing a shift register generator, we should avoid any duplicate use of shift register bit sequences generated in all parallel bit positions in the row direction or in the column direction. Simple criteria judging the absence of these duplication were derived in terms of the PFSR formalism. We have tested four generators in the literature and have found that two of them violate the criteria. Because a parallelized shift register generator has a very simple structure, there is a large chance that it reveals some regularity. We cannot be too careful in parallelizing a shift register generator.

Parallelization of random numbers in terms of equipartitioning is not peculiar to parallel computing. In many simulations on traditional serial computers, random numbers are distributed to definite degrees of freedom in a definite way so that they essentially form multiple parallel sequences. Statistical simulations of queues and physical simulations on lattices are good examples which may involve this problem. The results of our theoretical consideration would also be useful for the study of random numbers in these applications.

There are various types of parallel computers, each of which requires programs in its own style. We implemented the parallel generators on vector processors and distributed memory multiprocessors. Recently, vector-parallel machines which are multiprocessor systems with vectorial nodes have appeared. It is trivial that our generators are easily implemented on these machines. Application to shared memory multiprocessors would also be possible, but it may cause trouble in allocating a large memory space to the table of parallel random numbers. As we discussed in chapter 2, the same problem will occur on vector processors if the vector length is taken to be very long.

Throughout the work, we considered random numbers which behave as if they were independently sampled from the uniform distribution. Many applications, however, require random quantities simulating other kinds of distributions. For multiprocessors, there would be no problem to be discussed on parallel generation of these specific random quantities. In fact, these quantities are transformed from uniform random numbers, and conventional programs for such transformations work on each node processor. As for vector processors, however, some problems may be remained because these programs are also hoped to be vectorized.

In this work, no description was given on statistical tests of parallel random numbers. Presumably, testing of parallel random numbers may be the most important problem left for the future. For a single sequence, a series of statistical tests are used on trust to judge if the sequence is random enough for usual applications. We do not know what kind of statistical tests can judge the properties of parallelized random numbers. Applications to individual problems would take the place of these tests until powerful statistical tests are invented.

Acknowledgement

The author would like to express his sincerest thanks to Professor S. Shirahata for his warm support and encouragement given to the author. He is grateful to the members of Department of Applied Mathematics, Faculty of Engineering Science, Osaka University, for their warm hospitality at the department.

He wishes to acknowledge his grateful thanks to Professor K. Asai and Professor M. Sato for their interest and encouragement.

He thanks Professor O. Miyamura and Dr. T. Takaishi for collaboration. He also thanks Professor M. Fukui and Dr. S. Hioki for useful discussions.

Programs were tested on NEC SX-2N at the Computation Center of Osaka University, HITAC S820/80 at the Computer Center, University of Tokyo, Fujitsu AP1000 at Fujitsu Parallel Computing Research Facilities, and a personal computer DELL Latitude XP with Microsoft FORTRAN POWERSTATION compiler.

Bibliography

[1] S. Aluru, G.M. Prabhu and J. Gustafson, A random number generator for parallel computers, *Parallel Comput.* **18** (1992) 839-847.

[2] A.C. Davies, Delayed versions of maximal-length linear binary sequences, *Electron. Lett.*, **1** (1965) 61.

[3] A.M. Ferrenberg, D.P. Landau and Y.J. Wong, Monte Carlo Simulations: Hidden Errors from "Good" Random Number Generators, *Phys. Rev. Lett.* **69** (1992) 3382-3384.

[4] M. Fushimi and S. Tezuka, *Japanese J. Appl. Stat.*, **10** (1982) 151-163 (in Japanese).

[5] M. Fushimi, A reciprocity theorem on the random number generation based on m-sequences and its applications (in Japanese), *Transactions of the Information Processing Society of Japan* **24** (1983) 576-579.

[6] M. Fushimi and S. Tezuka, The k-distribution of generalized feedback shift register pseudorandom numbers, *Commun. ACM* **26** (1983) 516-523.

[7] M. Fushimi, Random number generation on parallel processors, in: E.A. MacNair, K.J. Musselman, P. Heidelberger, eds., *Proceedings of the 1989 Winter Simulation Conference*, (IEEE Press, New York, 1989) 459-461.

[8] S.W. Golomb, *Shift Register Sequences*, (Holden-Day, San Francisco, 1967).

[9] S.W. Golomb, *Shift Register Sequences*, revised ed. (Aegean Park Press, Laguna Hills, CA, 1982).

[10] J.P. Hayes and T. Mudge, Hypercube supercomputers, *Proc. IEEE*, **77** (1989) 1829-1841.

[11] N. Ito and Y. Kanada, Random number generation for the vector processor, *Supercomputer*, **7** (1990) 29-35.

[12] C. Kalle and S. Wansleben, Problems with the random number generator RANF implemented on the CDC Cyber 205, *Comput. Phys. Commun.*, **33** (1984) 343-346.

[13] S. Kirkpatrick and E.P. Stoll, A very fast shift-register sequence random number generator, *J. Comput. Phys.*, **40** (1981) 517-526.

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

[15] T.G. Lewis and W.H. Payne, Generalized feedback shift register pseudorandom number algorithm, *J. ACM* **20** (1973) 456-468.

[16] N.M. Maclaren, The generation of multiple independent sequences of pseudorandom numbers, *Appl. Statist.*, **38** (1989) 351-359.

[17] J. Makino and O. Miyamura, Generation of shift register random numbers on vector processors, *Comput. Phys. Commun.* **64** (1991) 363-368.

[18] J. Makino, T. Takaishi and O. Miyamura, Generation of shift register random numbers on distributed memory multiprocessors, *Comput. Phys. Commun.* **70** (1992) 495-500.

[19] J. Makino, On the structure of parallelized random number sources, *Comput. Phys. Commun.* **78** (1993) 105-112.

[20] J. Makino, Lagged-Fibonacci random number generators on parallel computers, *Parallel Comput.* **20** (1994) 1357-1367.

[21] J. Makino and O. Miyamura, Parallelized feedback shift register generators of pseudorandom numbers, *Parallel Comput.* **21** (1995) 1015-1028.

[22] G. Marsaglia, Random numbers fall mainly in the planes, *Proc. Nat. Acad. Sci.*, **61** (1968) 25-28.

[23] G. Marsaglia and T.A. Bray, One-line random number generators and their use in combinations, *Commun. ACM*, **11** (1968) 757-759.

[24] G. Marsaglia and L.H. Tsay, Matrices and the Structure of Random Number Sequences, *Linear Algebra and Its Applications* **67** (1985) 147-156.

[25] G. Marsaglia, A Current View of Random Number Generators, in: L. Billard, ed., *Computer Science and Statistics: Proc. 16th Symposium on the Interface* (North-Holland, Amsterdam, New York, 1985) 3-10.

[26] A. De Matteis and S. Pagnutti, Parallelization of random number generators and long-range correlations, *Numer. Math.*, **53** (1988) 595-608.

[27] J.C.P. Miller and D.J.S. Brown, An Algorithm for Evaluation of Remote Terms in a Linear Recurrence Sequence, *Comp. J.* **9** (1966) 188-190.

[28] J. Saarinen, J. Tomberg, L. Vehmanen and K. Kaski, VLSI implementation of Tausworthe random number generator for parallel processing environment, *IEE Proceedings-E* **138** (1991) 138-146.

[29] R.C. Tausworthe, Random numbers generated by linear recurrence modulo two, *Math. Comput.* **19** (1965) 201-209.

[30] A.N. van Luyn, Shift-register connections for delayed versions of m-sequences, *Electron. Lett.*, **14** (1978) 713.

[31] N. Zierler and J. Brillhart, On primitive trinomials (Mod 2), *Inform. Control*, **13** (1968) 541-554.

[32] N. Zierler and J. Brillhart, On primitive trinomials (Mod 2), II, *Inform. Control*, **14** (1969) 566-569.

[33] N. Zierler, Primitive trinomials whose degree is a Mersenne exponent, *Inform. Control*, **15** (1969) 67-69.