

FLORIDA STATE UNIVERSITY
COLLEGE OF ARTS AND SCIENCES

IMPLEMENTATION OF LEAP AHEAD FUNCTION FOR LINEAR
CONGRUENTIAL AND LAGGED FIBONACCI GENERATORS

By
NIRAJ PANDEY

A Project submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Summer Semester, 2008

The members of the Committee approve the Project of Niraj Pandey defended on August 6, 2008.

Dr. Michael Mascagni
Professor Directing Project

Dr. Ashok Srinivasan
Committee Member

Dr. Piyush Kumar
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

ACKNOWLEDGEMENTS

I would like to express my deep-felt gratitude to my advisor, Dr. Michael Mascagni of the Computer Science Department at The Florida State University, FL, for his advice, encouragement, enduring patience and constant support. Finally, I would like to thank my other comittee members, Dr. Ashok Srinivasan and Dr. Piyush Kumar for their help and support.

TABLE OF CONTENTS

List of Figures	vii
Abstract	viii
1. INTRODUCTION	1
1.1 Summary	3
2. Background	5
2.1 Introduction to Pseudorandom Numbers	5
2.2 Introduction to the Types of PRNGs	5
2.3 Parallel Random Number Generation	8
2.4 TestU01	10
2.5 Random Number Testing	10
2.6 Parallel Random Number Testing	14
3. Design and Implementation of Leap Ahead Functions	16
3.1 Motivation	16
3.2 Leap Ahead For Linear Congruential Generator (LCG)	17
3.3 Leap Ahead for Additive Lagged Fibonacci Generator (ALFG)	18
3.4 Leap ahead for Multiplicative Lagged Fibonacci Generator(MLFG)	29
3.5 Square and Multiplication Algorithm	30
4. Parallel Random Number Testing	32
4.1 PRNG Tests	32
4.2 Design of Parallel RNT	33
4.3 Performance and Result	35
5. Concluding Remarks	37
5.1 Conclusion	37
5.2 Future Work	38
A. Appendix A	39
A.1 Introduction to TestU01	39
A.2 PRNG tests in modified crush	40
A.3 Introduction to MPI	46

B. Appendix B 48
 B.1 LCG Leap Ahead 48

C. Appendix C 52
 C.1 Matrix Based ALFG Leap Ahead 52
 C.2 Polynomial Based ALFG Leap Ahead 62

REFERENCES 75

LIST OF FIGURES

- 3.1 The output of the matrix based ALFG algorithm 22
- 3.2 The output of the TestU01 ALFG generator 23
- 3.3 The output of the polynomial based ALFG algorithm 28

ABSTRACT

This report presents leap ahead functions that can be used to jump ahead an arbitrary amount in the period of linear congruential generators (LCGs), additive lagged-fibonacci generators (ALFGs) and multiplicative lagged-fibonacci generators (MLFGs). Leap ahead functionality of these pseudorandom number generators (PRNGs) is desirable in many situations. One application of the leap ahead functions is to test PRNGs in parallel on a cluster environment. Besides this, the leap ahead functions can be used independently for other purposes as well as mentioned in [1].

Leap ahead functions for ALFGs and MLFGs are implemented using the both polynomial and matrix method [1]. The Leap ahead functions thus implemented were later used to test above mentioned PRNGs in parallel against different random number test (RNT) routines implemented in *TestU01*. Testing PRNGs is a computationally intensive task. Leap ahead functions can be useful to speed up the process of RNT by splitting the random number stream obtained from the sequential PRNGs into several substream and then testing each of these substreams independently against RNT routines in separate cluster node. This use of leap ahead function to test sequential PRNGs in a parallel manner was conducted as part of this project to verify the correctness and demonstrate the use of leap ahead functions.

CHAPTER 1

INTRODUCTION

Random number generators are widely used in many computational science and engineering fields. For example: simulation, sampling, numerical analysis, computer programming, decision making and so on. True random number generators are normally implemented through a physical device that plugs into computer and produces genuine random numbers as opposed to the pseudorandom numbers that are produced by a computer program. Random number sequence generated by computer programs are called pseudorandom. Pseudorandom numbers generated by deterministic algorithms are very close to a true random number, but never can be considered truly random. There are various applications whose outcome is hugely dependent on the quality of random number generated. Therefore, before using pseudorandom numbers, each PRNG should be fully tested to verify that their statistical properties resembles that of true random numbers. Depending upon the requirement of an application, random numbers are generated either in a sequential or parallel fashion. Sequential random number generators produce all the random numbers in a single node. whereas parallel random number generators produce streams of random numbers on a different nodes. Normally, parallel random number generation is a complicated process. Caution should be taken to avoid inter-stream correlations.

Ideally sequential PRNG would produce a stream of numbers that have following properties:

- uniformly distributed.
- uncorrelated.
- never repeats itself.

- satisfy any statistical test for randomness.
- reproducible.
- portable.
- can be changed by adjusting an initial ‘seed’ value.
- can be easily be split into many independent subsequences.
- can be generated rapidly using limited computer memory.

Practically, it is impossible to satisfy all these properties. For practical purpose, the period of repetition of the sequence must be much larger than the number of pseudo random number that might be used in any application and that the correlations be small enough so that they do not noticeably affect the outcome of a computation.

Often the speed of random number generators has been the cause for the low quality of random number produced. Users who want super-fast random number generator (RNG) usually have applications that spend most of their time generating random numbers, and require a huge number of them. These types of applications are often the ones that are most sensitive to the quality of the generator, in which case it would seem prudent to sacrifice a little speed for much better randomness properties. There are various ways of speeding up the random number generation, one of which is to utilize the leap ahead function to generate random numbers in parallel. Ideally parallel PRNGs would produce a stream of numbers that have following properties:

- should work for any number of processors.
- each processor should satisfy all the requirements of a good sequential generator, e.g. they should be uniformly distributed, uncorrelated, and have a large enough period.
- no correlations between the sequences on different processors.
- same sequence of random numbers should be produced for different numbers of processors and for the special case of a single processor.

- algorithm should be efficient, i.e. after the generator is initialized, each processor should generate its sequence independently of the other processors.

All these requirements should be met by a good parallel random number generator.

To determine the quality of a given random number generator, random number tests are used. An obvious requirement for a good parallel random number generator is that the sequential generator on which it is based should have acceptable randomness properties. The many standard statistical tests for checking the randomness properties of sequential generators can be applied to parallel generators, by testing the random number streams on each processor, and from all processors combined. This is the usual approach in testing parallel generators. Since the parallel random number generator should also be a good sequential random number generator. The idea of testing a parallel random number generator on each node of a cluster can be used to parallelize the testing procedure for sequential random number generator; However, for testing sequential random number generator in a parallel manner, it requires leap ahead function. This was the motivation for this project. In this project, we have implemented leap ahead function for LCG and lagged Fibonacci generators. Leap ahead functions can jump an arbitrary amount in the period of PRNGs. This property of the leap ahead function can be used to test their respective sequential PRNGs in a parallel manner. Ability to Leap ahead in the period of a PRNG is a very important feature because the parallel RNT requires each node on a cluster to test random numbers from certain range within the period of the PRNGs. This implies that the RNT routine running on each node has to first jump ahead an amount that is equal to the lower bound of its range and secondly start testing numbers within its range.

1.1 Summary

The rest of this report is organized as follows. Chapter 2 serves as a background for understanding the underlying concept of this project. Chapter 3 describes the detail design and implementation of this project as well as the motivation for the implementation of leap ahead functions. Chapter 4 describes one of the many uses of leap ahead functions. As part of verifying the correctness of leap ahead functions, this chapter describes the procedure and result of testing sequential LCG and ALFG using RNT routines available in *TestU01*.

Chapter 5 provides some concluding thoughts and mentions some future works. Finally, appendix A lists some of the tools and concept used in this project and appendix B and C lists source code for the leap ahead functions.

CHAPTER 2

Background

2.1 Introduction to Pseudorandom Numbers

Random numbers are used everywhere from casino machines to scientific applications such as statistical sampling, Monte Carlo methods, and computer security. There are basically two types of sources for the generation of random numbers, one is the physical source whereas the other is the computer algorithm. Random numbers generated from the physical sources are also called true random numbers. Physical sources can be thermal noise or the photoelectric effect or other quantum phenomena. A special hardware device that plugs into computer is required to interface between the computer software and the physical source of random numbers. Random numbers generated from the computer algorithms are also called pseudorandom numbers. Random numbers are produced from each of these sources using completely different approaches. A computer can only generate true random numbers through some physical sources. Unfortunately, hardware random number generators are seldom used because they are expensive, slow, not reproducible, and often not as random as simple PRNGs. Pseudorandom number generation is another way of producing random numbers in which random numbers are generated by a deterministic algorithm based on some mathematical recurrence relation. When a PRNG is carefully designed, it can produce random numbers with many desirable properties.

2.2 Introduction to the Types of PRNGs

In this project we are interested in the implementation of the leap ahead function for LCGs, ALFGs and MLFGs. We use these leap ahead functions to test the respective PRNGs in parallel against some RNT routines implemented in *TestU01*. A good random number

generators should generate streams of numbers that are random, uniformly distributed, portable, reproducible, homogeneous, and have long periods. In addition, these generators should also be able to produce random numbers quickly and efficiently. These attractive properties allow PRNGs to be considered the most suitable type of random number generator in many applications. The random and uniformly distributed properties suggest that while we should not be able to detect any patterns in a set of random numbers, these numbers should also be spread out evenly. Portable means that a random number generator should produce the same results on different computer platforms. For example, a generator should produce the same set of numbers on any 32-bit machine and on any 64-bit machine. Reproducibility suggests that we can get the same sets of random numbers no matter how many times we use the same random number generator, if desired. Homogeneous implies that all bits of a number are equally likely to change and hence are all equally random. Finally, the period of a generator is the length of a stream of random numbers before these numbers start repeating. This length should be as long as possible and at least several times larger or the square of the total of the random numbers that are required.

There are many different types of random number generators. In this project we are interested in only three types of random number generators namely LCG, ALFG and MLFG.

1. *Linear Congruential Generator:*

Probably the most commonly-used random number generators are LCGs. LCGs produce a sequence of X_i of random number using the relation

$$X_i = (a * X_{i-1} + c) \pmod{M}, \tag{2.1}$$

where a is the multiplier, M is the modulus, and c is an additive constant. The parameters (a, c, M) must be chosen carefully to ensure a large period, good uniformity and randomness properties. The maximum period of M is possible if c is relatively prime to M , $a - 1$ is a multiple of p , for every prime p dividing M and $a - 1$ is a multiple of 4, if M is a multiple of 4 [2].

- *Combined Linear Congruential Generator:* Different streams of random numbers generated by using LCGs as $(X_n^{(j)})_{n \geq 0}, 1 \leq j \leq r$, can be combined into a new

stream $(X_n)_{n \geq 0}$, $X_n \equiv X_n^{(1)} + \dots + X_n^{(r)} \pmod{1}$, that yields an easy way to achieve long periods while keeping the computational cost of generating the number low by choosing suitable parameters for each underlying generators.

This project provides an interface to combine number of LCGs and use leap ahead function to compute j th ($j > 0$) element in the period of such PRNGs. LCGs are combined using the following relation:

$$X_{1,i} = (a_1 \times X_{1,i-1}) \pmod{m_1}$$

$$X_{2,i} = (a_2 \times X_{2,i-1}) \pmod{m_2}$$

$$X_{3,i} = (a_3 \times X_{3,i-1}) \pmod{m_3}$$

$$X_{4,i} = (a_4 \times X_{4,i-1}) \pmod{m_4}$$

$$U_i = \left(\frac{X_{1,i}}{m_1} + \frac{X_{2,i}}{m_2} + \frac{X_{3,i}}{m_3} + \frac{X_{4,i}}{m_4} \right) \pmod{1.0} \quad (2.2)$$

Where a_j , m_j are the multiplier and the modulus for the LCG X_j , $1 \leq j \leq 4$. This generates the pseudorandom number U_i , which is uniformly distributed over the interval $[0, 1)$.

2. Lagged Fibonacci Generators

Lagged fibonacci-generators are the generalization of the Fibonacci sequence. These generators can produce astonishingly long period of random numbers. Each element in a sequence is defined as follows:

$$X_i = X_{i-p} \odot X_{i-q} \pmod{M}, \quad p > q \quad (2.3)$$

where p is called the register length, q is called the lag, M is the modulus and \odot can be any binary arithmetic operations $(+, -, \times)$. In this project we are interested in the implementation of leap ahead function for additive $(+)$ and multiplicative (\times) lagged-fibonacci generators.

- *Additive Lagged-Fibonacci Generators (ALFG)*

$$X_i = X_{i-p} + X_{i-q} \pmod{M}, \quad p > q \quad (2.4)$$

When M is prime, period is as large as $M^p - 1$. However, it is more common to consider lagged-fibonacci generators with $M = 2^m$, for some $m > 0$. In later case, the maximum possible period is $(2^p - 1)2^{m-1}$ [3, 4]. ALFGs with power-of-two moduli are considerably easier to implement than general prime moduli; however, their periods are much smaller than in the prime-modulus case [3, 4]. This maximal period of the ALFG is reached if and only if the characteristic polynomial $f(x) = x^p - x^{p-q} - 1$ is a primitive polynomial modulo 2 [2].

- *Multiplicative Lagged Fibonacci Generators (MLFG)*

$$X_i = X_{i-p} \times X_{i-q} \pmod{M}, \quad p > q \quad (2.5)$$

This generator is defined by the modulus, M the register length, p , and the lag, q [5, 4]. It is more common to consider lagged-fibonacci generators with $M = 2^m$, for some $m > 0$. The maximum possible period is $(2^{m-3}(2^p-1))$. This maximal period is reached if and only if the characteristic polynomial $f(x) = x^p - x^{p-q} - 1$ is a primitive polynomial modulo 2 [2].

2.3 Parallel Random Number Generation

Monte Carlo simulations have always been one of the main tasks for the most advanced computers of the age. In these days, the advent of highly parallel machines with supreme performance urges efforts to develop various techniques of handling Monte Carlo Simulations in parallel environments. Correlations in the pseudorandom number sequence can lead to errors in the results of MC computations. In parallel Monte Carlo applications, each process generates a distinct stream of pseudorandom numbers. This might cause not only intra-stream correlations but also inter-stream correlations. In view of these problems with parallel random number generations, there are various ways to parallelly generate random numbers.

1. *Parallelization Through Random Seed*

In this parallelization scheme, each processor chooses a start state randomly and hope that the seeds will take them to widely separated portions of the sequence, so that there will be no overlap between the sub-sequence used by different processors [5].

2. *Parallelization Through Blocking*

In this parallelization scheme, the user determines a block size, B . if the PRNG sequence is given by x_0, x_1, \dots , then the parallel PRNG sequence on processor i is given by X_{iB}, X_{iB+1}, \dots . The danger here is that if the user happens to consume more random numbers than expected, then the stream on different processors could overlap[5].

3. *Parallelization Through Leap-Frog*

In the leap frog scheme, processor i gets the sequence $x_i, x_{i+p}, x_{i+2p}, \dots$, where P is the total number of processors. Long range correlations in the PRNG become short-range intra-stream correlations in parallel PRNG, which can be worse than inter-stream correlations[5].

4. *Parallelization Through Parameterization*

In this parallelization scheme, each processor gets independent, full-period sequence. This scheme makes reproducibility of random numbers easy when it is desired. Parameterization of PRNGs is based on the fact that given a sequence number i , there is an easy way of generating the i th sequence. This type of parameterization can be done in two ways. *Seed parameterization* is used with certain generators where the set of possible initial states of the PRNG naturally divides itself into a number of smaller cycles, where each seed from different cycle can be given to different processors. In *iteration function parameterization*, a different iteration function is used for each stream. In order to achieve this, we need a way of parameterizing the iteration function so that given i , the i th iteration function can be produced easily. PRNGs is based on the fact that given a sequence number i , there is an easy way of generating the i th sequence. This type of parameterization can be done in two ways. *Seed parameterization* is used with certain generators where the set of possible initial states of a PRNG naturally

divides itself into a number of smaller cycles, where each seed from a different cycle can be given to different processors. In *iteration function parameterization*, a different iteration function is used for each stream. In order to achieve this, we need a way of parameterizing the iteration function so that given i , the i th iteration function can be produced easily. [5]

2.4 TestU01

TestU01 is a software library implemented in the ANSI C language. It provides different utilities for the empirical statistical testing of PRNGs. It provides general implementation for the most of the classical statistical tests for PRNGs along with other proposed in the literature. Three predefined batteries of tests namely *small crush*, *crush* and *big crush* for the sequence of random numbers over the interval (0,1) and for bit sequences are available. Various tools to perform systematic studies of the interaction between a specific test and the structure of the point sets produced by a given family of PRNGs are also offered. For e.g. given a kind of test and class of PRNGs, there are tools implemented in *TestU01*, that can be used to determine how large should be the sample size of the test, as a function of the generator's period length, before the generator starts to fail the test systematically. Besides these, the library also provides the generic implementation for various types of PRNGs along with specific PRNGs that are found in widely used software or read literature [6]. Various statistical tests available in *TestU01* can be applied to the generators already implemented in the library or to the user-defined generators.

2.5 Random Number Testing

Before using the PRNG on any application, thorough testing is required to ensure that the generator possess all the required properties of a good PRNG. Two types of random number tests are popular: *empirical tests*, for which the computer program manipulates groups of number of the sequence and evaluates certain statistics and *theoretical tests*, for which we establish characteristics of the sequence by using number-theoretic methods based on the recurrence rule used to form the sequence [2]. Stastical tests of a PRNG typically compute some statistic from a portion of a random stream. This statistic is compared against the expected value from an uniformly distributed truly random sample. If the results from

the PRNGs are consistent with those from a random sample, then the test is said to have been "passed" by the random number generator. Passing a test does not imply that the generator is producing a truly random sequence. It just means that the particular test could not differentiate between a truly random stream and the stream generated by the generator under consideration. However, if several different tests are passed, then our confidence in the random number generator increases [2].

In this project we have implemented leap ahead functions that can be used to test PRNGs in parallel. There are many tests which are designed to test particular statistical property of a random number sequence generated by a PRNG. For instance, sequential PRNGs should be tested for intra-stream correlations. This project uses some of the most popular empirical tests from *Knuth* and *emphMarsaglia* implemented in *TestU01* to test LCG, ALFG and MLFG using the respective leap ahead functions.

Here are the descriptions of those tests from *TestU01*, that are used in this project to demonstrate the use of leap ahead functions to test sequential PRNGs in parallel.

1. *Frequency Test*

Numbers in random number sequence should be uniformly distributed over the range $(0, d - 1)$. In Frequency Test, either we use:

- KS test with $F(x) = x$ for $0 \leq x \leq 1$.
- let d be a convenient number, such as 64 or 128 and use this sequence instead of above. For each integer r , $0 \leq r < d$, count the number of times that $Y_j = r$, where Y_j is an integer, that is independently and uniformly distributed between 0 and $d - 1$, for $0 \leq j < n$ and then apply the chi-square test [2].

2. *Serial Test*

Serial test tests whether pair of successive numbers are uniformly distributed in an independent manner. To carry out the serial test, we count the number of times that the pair $(Y_{2j}, Y_{2j+1}) = (q, r)$ occurs, for $0 \leq j < n$. These counts are made for each pair of integers (q, r) with $0 \leq q, r < d$, and the χ^2 test is applied to these $k = d^2$ categories with probability $1/d^2$ in each category [2].

3. *Gap Test*

This test examines the length of “gaps” between occurrences of random numbers. For e.g., if $[a, b] = [0.4, 0.7]$ and the sequence is $0.1, 0.5, 0.6, 0.9, \dots$, then the length of the first gap (between number 0.5 and 0.6) is 2. In this test, n such gaps are recorded and lump gap lengths greater than t (t is the gap length) is analyzed [2].

4. *Poker test (Partition test)*

We Generate k integers in $[0, d - 1]$ and count the number of distinct integers obtained. For e.g., if $k = 3$ and $d = 3$ and the sequence is $: 0, 1, 1, \dots$, then the number of distinct integers obtained in the first 3-tuple is 2. We repeat this process for n times and compare with the expected distribution for random samples from the uniform distribution. [2]

5. *Coupon Collector's test*

Suppose we have a set of random number in the range $[0, 2^{32} - 1]$. We linearly search through the set and produce n numbers, where each n is the length of the smallest subset containing all $2^{32} - 1$ possible random numbers. We compare the observed n which the corresponding expected value [2].

6. *Permutation test*

Suppose the n subsequences with t random numbers in each subsequence constitute the entire random number sequence. Each subsequence should have $t!$ possible permutations. We count the frequency of each permutation and then apply the Chi Square test. [2]

7. *Run Test*

We take a random number sequence and count the lengths that consecutive random numbers are monotonically increasing to get n lengths. Then we test these actual lengths with the expected lengths. [2]

8. *Maximum-of-t test*

Maximum of t tests for the biggest numbers in n random real number sequences in the range $[0, 1)$. This number should have the distribution x^t , where t is length of each sequence [2].

9. *Collision test*

The Collisions test involves first generating a series of new integers, and then test the series based on collisions. The $\log d$ most significant bits from $\log md$ random integers can be used to concatenate into a new integer. We use $n \log md$ random integers to form n concatenated numbers, where n should be significantly smaller than m . Next, we count the frequencies for numbers that appear more than once in the set of new numbers. Finally, we compare the observed frequencies and expected frequencies of these numbers. [2]

10. *Birthday spacing test*

This test was introduced by George Marsaglia. Suppose the birthdays are (Y_1, \dots, Y_n) , where $0 \leq Y_k < m$ (where we can think of m as “days of year” and n as “birthdays”). Sort them into nondecreasing order $Y_{(1)} \leq \dots \leq Y_{(n)}$, then define n “spacings” $S_1 = Y_{(2)} - Y_{(1)}, \dots, S_{n-1} = Y_{(n)} - Y_{(n-1)}, S_n = Y_{(1)} + m - Y_{(n)}$; finally sort the spacings into order, $S_{(1)} \leq \dots \leq S_{(n)}$. Let R be the number of equal spacings. We repeat the test with certain m and n values for x times and do a χ^2 test to compare the empirical R 's with the correct distribution. [6]

11. *Marsaglia Serial Over Test*

Implements the overlapping t -tuple test. It is similar to the Knuth's serial test, except that the n vectors are generated with overlap, as follows. A sequence of uniforms u_0, \dots, u_{n-1} is generated, and the n points are defined as $(u_0, \dots, u_{t-1}), (u_1, \dots, u_t), \dots, (u_{n-1}, u_n, u_0, \dots, u_{t-3}), (u_n, u_0, \dots, u_{t-2})$. [6]

12. *Marsaglia Collision Over Test*

This is similar to collision test, except that the vectors are generated with overlap. If n (the number of points) and d^t (the number of cells) are very large and have the same order of magnitude, then the number of collisions C is a random variable which is approximately normally distributed with mean $\mu \approx d^r(\lambda - 1 + e^{-\lambda})$ where $\lambda = (n - t + 1)/d^t$ and variance $\sigma^2 \approx d^t e^{-\lambda}(1 - 3e^{-\lambda})$. [6]

2.6 Parallel Random Number Testing

With the growing use of random numbers on various fields, new random number generators are frequently developed and implemented. Before using any of these PRNGs, one needs to test these generators thoroughly to verify that the quality of random number produced by these generators are genuine. Often, these generators need to be tested for very large sequence of random numbers. With the advent of parallel computing and the availability of more memory and processing power, hardware constraint that existed in past is no more. Only issue remaining is the development of an efficient algorithm that could harness the parallel computing capability provided by the hardware. In recent days, there has been significant progress in the design and development of parallel PRNGs and RNTs. One such example is a *SPRNG* library [7].

In this project we are also interested to use the leap ahead functions to test the PRNGs in parallel. Our approach is to split the long sequence of the random numbers generated by a sequential PRNG into various sub-streams and test each of these sub-streams on a node in a cluster environment . To split the long stream of random numbers, we use leap ahead function. Leap ahead functions can jump ahead an arbitrary amount in the period of a PRNG without ever producing the intermediate random numbers. This feature can be utilized to seed the PRNG from any value of our interest within the period of the PRNG. In this way, we can generate multiple sub-streams of random numbers and use RNT to test each of these sub-streams on a node. For e.g. if we want to test 100 random numbers and 10 nodes are available, we could split the main stream of random numbers into 10 sub-streams using the leap ahead function. Leap ahead function running in nodes 0, 1, 2, ..., 8, 9 will jump ahead 0, 10, 20, ..., 80, 90 amount respectively in the period of the PRNG of our interest. The PRNG's state value obtained from the leap ahead function in each node will be used to seed the copy of PRNG running in that node. Once the process of testing random numbers in

each node is completed, its result will be sent to the main node. Main node will gather the results from all the nodes and computes the test statistics. This way of using leap ahead function to test the sequential PRNG in parallel can greatly reduce the computation time when the sequence of random numbers being tested is very large.

For certain classes of PRNGs, it is very difficult to devise leap ahead function. Normally, it is easier to formulate a leap ahead function for PRNGs based on linear recurrence rather than PRNGs based on non-linear recurrence relation.

CHAPTER 3

Design and Implementation of Leap Ahead Functions

3.1 Motivation

Linear recurrence sequences with very large periods are widely used as the basic building block for the PRNGs. In many simulation applications, multiple streams of the random numbers are needed. There are various uses of the multiple streams of random numbers provided that each of these sub-streams are independent and identically distributed (IID). In this project, we consider the use of leap ahead function to split the stream of random numbers generated from the sequential PRNG to obtain the multiple sub-streams as described in section 2.6. There are several situations where the ability to leap ahead an arbitrary amount in the period of a pseudorandom number generator is desirable [8]. A simple situation occurs when the same computation using a PRNG is required to be performed multiple times with different set of random numbers [1]. Normally, a user can provide a different seed for each of the sub-streams and this seed situates the user's computation on a different starting value in the period of a PRNG [1]. This method only guarantees that the new starting values are different from the old one. However, there might be a huge overlap between the first and second set of random numbers. One obvious solution to this problem is to use leap ahead function that takes an offset in the PRNG's period as an argument and starts the new stream at an offset away from the old starting value [1]. This concept can be applied to test PRNGs in parallel, that involves splitting a long stream of random numbers into multiple sub-streams. This basic idea of being able to produce multiple sub-streams of random numbers by splitting a contiguous sequence of random numbers is the main motivation for this project. In this project we have not only implemented the leap ahead functions but are also used these leap ahead functions to test sequential PRNGs in parallel.

While testing a sequential PRNG with RNTs that consume billions of random numbers,

it takes hours of computation time. This time can be significantly reduced by splitting the main stream of PRNGs into several sub-streams and testing each of these sub-streams of random numbers on a separate node. This process of splitting a long stream of random numbers into several sub-streams requires an efficient and fast leap ahead function for the particular PRNG of interest. A fast leap ahead function enables one to jump ahead an arbitrary amount "j" in the period of PRNG in $O(\log_2 j)$ "operations" [1]. Each nodes in a cluster has to test its own range of random numbers. To start the generation of random numbers beginning at its range (say $n - n + j$) requires jumping ahead n in the period of PRNG. While the subsequent node is required to jump $n + j$.

This project implements an efficient and fast leap ahead function for LCG, ALFG and MLFG. An algorithm for a fast leap ahead function for LCG is based on the generalised LCG formula outlined by Knuth [2]. Similarly, an algorithm for a fast leap ahead function for ALFG and MLFG is based on the method described by Mascagni in his famous paper [1]. These leap ahead functions are later used to test PRNGs based on LCG, ALFG and MLFG, that are implemented in *TestU01* using the RNT routines available in *TestU01*. Leap ahead functions for the ALFG and MLFG is implemented using two methods based on the matrix and the polynomial representation. MLFG uses the leap ahead functions of ALFG to implement its leap ahead function.

3.2 Leap Ahead For Linear Congruential Generator (LCG)

The LCG is the most commonly used generator for pseudorandom numbers. It was introduced by D. H. Lehmer in 1949 [2] and is based on the following recursion:

$$X_{n+1} = a * X_n + c \pmod{M}, \tag{3.1}$$

where a is the multiplier, M is the modulus, and c is an additive constant. The parameters (a, c, M) must be chosen carefully to ensure a large period, good uniformity and randomness properties. The maximum period of M is possible if c is relatively prime to M , $a - 1$ is a multiple of p , for every prime p dividing M and $a - 1$ is a multiple of 4, if M is a multiple of 4 [2].

$$X_{n+k} = (a^k X_n + (a^k - 1) \times c/b) \pmod{M}, \quad k \geq 0, n \geq 0, b = a - 1 \quad (3.2)$$

This expresses the $(n + k)$ th term directly in terms of the n th term. When $n = 0$, X_0 is the starting seed. It follows that the sequence consisting of every k th term of $\langle X_n \rangle$ is another linear congruential sequence, having the multiplier $a^k \pmod{M}$ and the increment $((a^k - 1)c/b \pmod{M}$ [2].

3.2.1 LCG Leap Ahead Implementation

The leap ahead function for LCG is based on the equation (3.2) [2]. This involves the calculation of a^k using the square and multiplication (SQM) algorithm outlined in section 3.5 [1]. The SQM algorithm reduces the number of modular multiplication to compute a^k from k to $O(\log_2 k)$. The naive approach for computing a^k involves k modular multiplication of a .

Following is the C interface provided in this project for the LCG leap ahead function:

```
typedef unsigned long Long;
Long mod_mul(Long base, Long exp, Long mod);
```

This function returns $base^{exp} \pmod{mod}$ using SQM algorithm 3.5.

```
Long lcg_leap_ahead(Long x0, Long a, Long c, Long mod, Long n);
```

This function returns the n th period value from the LCG as defined by equation (3.1), where x_0 is the starting seed, a is the multiplier, mod is the modulus, c is an additive constant and n is the leap ahead number passed as the parameter to the function.

3.3 Leap Ahead for Additive Lagged Fibonacci Generator (ALFG)

The recurrence relation for the ALFG is defined by the equation (2.4). Since it is a linear recurrence relation, a leap ahead function can be easily devised. This project implements the leap ahead function for the ALFG based on the matrix and the polynomial representation as outlined in [1].

3.3.1 Matrix Method

Given an initial vector $x_n = [x_0, x_{-1}, \dots, x_{-p-1}]^T$ as the starting seed values for the ALFG, the recursion can be defined by using the vector as follows:

$$x_n = Ax_{n-1} \pmod{M} \quad (3.3)$$

where M is 2^m . Matrix A is given by the following template:

$$A = \begin{bmatrix} & & & & & q & & & & p \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 & 0 & 1 \\ 1 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 1 & 0 \end{bmatrix}$$

This is a $p \times p$ matrix with elements defined modulo $M = 2^m$. Given the initial seed vector x_0 , we can compute j th vector in the sequence defined by the equation (3.3) as

$$x_j = A^j x_0 \pmod{M} \quad (3.4)$$

Algorithm For Matrix Based Leap Ahead

Input: the initial state vector $s = s_0$

Output: the j th state vector s_j

Setup the initial matrix A as shown in section 3.3.1

Compute the matrix A^j by the multiplication of initial matrix A into j times. Use SQM algorithm as outlined in section 3.5 [1] for the multiplication.

Compute the product of $A^j \times s_0$ to get the j th vector s_j

Return s_j

Algorithm 1: Matrix based leap ahead for ALFG

Efficiency of The Algorithm Computation of A^j can be accomplished with $O(\log_2 j)$ matrix-matrix multiplication with the well-known SQM algorithm [1]. Computation of $A^j \times x_0$ requires a single matrix-vector multiplication. The cost for matrix-vector multiplication for the matrix of size $p \times p$ and the vector of size p is p^2 multiplications and p additions. Matrix-matrix multiplication for the matrices of size $p \times p$ with no special structure requires

p^3 multiplications and $p^3 - p^2$ additions. The overall complexity of the matrix based leap ahead is $p^3 \log_2 j$. The storage requirement is $3p^2$ i.e. 3 matrices are required, one to store the current iterate, second to store the matrix A and third to store the temporary result. This memory requirement imposes a constraint on leap ahead for larger lags (p) like 1279×1279 . In such a scenario, the polynomial based leap ahead function defined in section 3.3.2 is more effective. Note that multiplication and the addition arithmetic operations discussed above are modulo M .

Interface for the matrix based leap ahead Following are the list of important functions that are implemented for the matrix based leap ahead function. Complete list of source code is available in appendix.

```
int init_matrix(int p, int q, unsigned long ***template, unsigned long ***ident, unsigned long ***temp);
```

This function initializes 3 matrices that are required for the function, which implements the SQM algorithm 3.5.

```
void sqm(unsigned long leap, unsigned long M, unsigned long **template_matrix, unsigned long **ident_matrix, unsigned long **temp_matrix);
```

This function multiplies the template matrix (*template_matrix*) as defined by the equation (3.3) into *leap* times using the SQM algorithm outlined in section 3.5. Variables M is the modulus, *ident_matrix* is the identity matrix to begin the multiplication process. During the SQM process, it keeps on storing the intermediate results too. The Other variable *temp_matrix* is used as a temporary storage during the multiplication process.

```
void vector_mult(unsigned long M, unsigned long *seed, unsigned long **ident_matrix, unsigned long **temp_matrix);
```

This function computes the product of matrix and vector modulo multiplication. Variables M is the modulus, *seed* is the initial seed vector, *ident_matrix* is the one that is multiplied with *seed* vector and the *temp_matrix* is a temporary storage used during the multiplication process.

```
void run_plfg(unsigned long leap, unsigned long M, int lag, int k, unsigned long **template_matrix, unsigned long **ident_matrix, unsigned long **temp_matrix, unsigned long **seed);
```

This is a wrapper function, that calls all the necessary functions to compute the vector after leaping ahead certain amount in the period of the ALFG. Parameter *leap* is the leap ahead number, *M* is the modulus value, *lag* is the larger lag value, *k* is the smaller lag value, *template_matrix* is used to store the matrix *A* of the equation (3.3) and *ident_matrix* and *temp_matrix* are used by the function *sqm* as explained in the paragraph 3.3.1.

Correctness of The Algorithm The implementation of the matrix based leap ahead algorithm was tested in two ways. First method is called as the *unit testing*. Using the concept of *unit testing*, each function was tested independently of the other functions. For instance, the major component of this algorithm was SQM. It was tested with matrices of various sizes.

Second method was used to test the overall output of the algorithm with some standard ALFG generator. We used the *TestU01* library's ALFG generator to compare the state of the generator given by the output of the matrix based leap ahead algorithm. The output of the matrix based leap ahead algorithm after leaping *n* number ahead was compared with the state of the *TestU01* ALFG generator after generating *n* number of random numbers. We found the indential output for extensive number of inputs. Here is an instance of the various tests conducted to verify the correctness of the algorithm. The parameters for the ALFG generator as defined in the equation (2.4) were $p = 5$, $k = 3$, $M = 2^{13}$ and the leap ahead number as defined in the equation (3.4) was $j = 3278456$. Following is a snippet of code to call matrix leap ahead implementation.

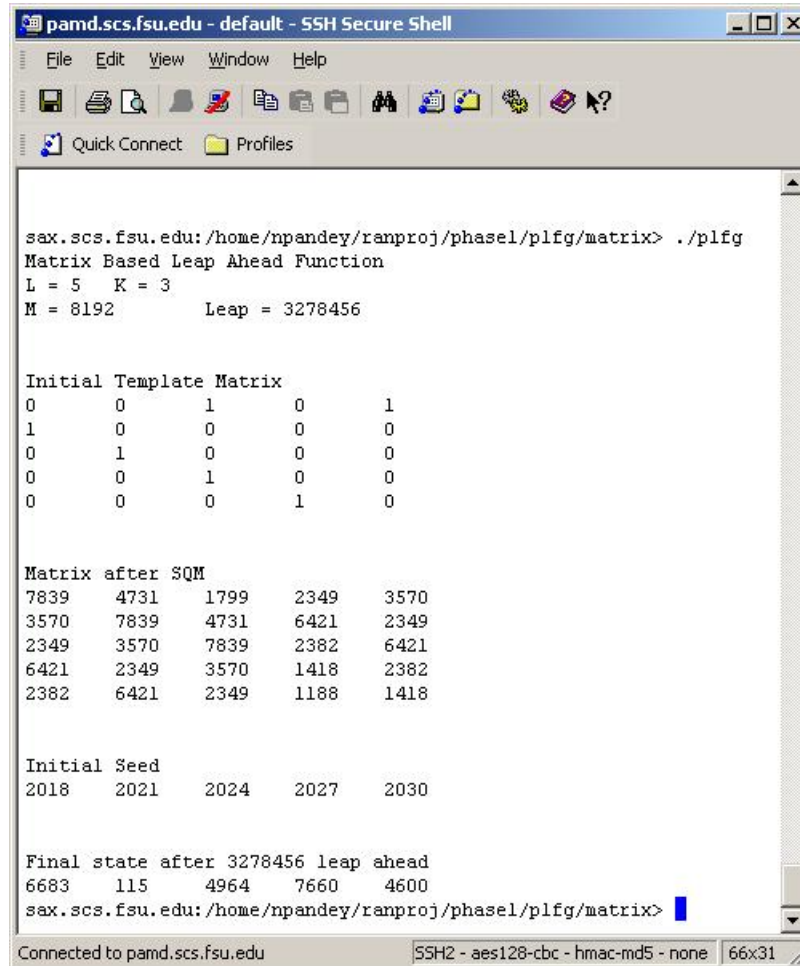
```
#include "plfg_utils.h"

int main() {
    Long **template_matrix, **ident_matrix, **temp_matrix, *
        seed;
    Long M = 13, j = 3278456;
    int p = 5, q = 3;

    M = 1 << M;
    run_plfg(j, M, p, q, template_matrix, ident_matrix,
        temp_matrix, seed);

    return 0;
}
}
```

Here is the screenshot of the output of the above program.



```
pamd.scs.fsu.edu - default - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles

sax.scs.fsu.edu: /home/npandey/ranproj/phasel/plfg/matrix> ./plfg
Matrix Based Leap Ahead Function
L = 5   K = 3
M = 8192   Leap = 3278456

Initial Template Matrix
0  0  1  0  1
1  0  0  0  0
0  1  0  0  0
0  0  1  0  0
0  0  0  1  0

Matrix after SQM
7839  4731  1799  2349  3570
3570  7839  4731  6421  2349
2349  3570  7839  2382  6421
6421  2349  3570  1418  2382
2382  6421  2349  1188  1418

Initial Seed
2018  2021  2024  2027  2030

Final state after 3278456 leap ahead
6683  115  4964  7660  4600
sax.scs.fsu.edu: /home/npandey/ranproj/phasel/plfg/matrix> █

Connected to pamd.scs.fsu.edu   SSH2 - aes128-cbc - hmac-md5 - none   66x31
```

Figure 3.1: The output of the matrix based ALFG algorithm

Similarly, the following snippet of code can be executed to run the ALFG generator implemented in the *TestU01* library. Note that *TestU01* library should be installed before running this code. For further instructions on how to install and use *TestU01* library, refer the document [6].

```
#include <stdio.h>
#include "unif01.h"
#include "umrg.h"
int main() {
    int p = 5, k = 3, r;
    unsigned long seed[p], z, M = 13, j = 3278456, i;
```

```

unif01_Gen *gen;

r = 2018;
M = 1 << 13;
for(i = 0; i < p; i++, r += 3)
    seed[i] = r;
gen = umrg_CreateLagFib(M, p, q, '+', 0, seed);
printf("\%s\n", gen->name);
unif01_WrLongStateFlag = 1;
unif01_WriteState(gen);

for(i = 0; i < j; i++)
    gen->GetBits(gen->param, gen->state);
unif01_WriteState(gen);

return 0;
}

```

Here is the screenshot of the output of the above program.

```

pamd.scs.fsu.edu - default - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
sax.scs.fsu.edu: /home/npandey/ranproj> ./mod_pow
umrg_CreateLagFib:  t = 13,  k = 5,  r = 3,  Op = +,  Lux = 0,  S = {2018, 2021, 2024, 2027, 2030}

Generator state:
S = {
    2018,      2021,      2024,      2027,      2030
}

After the generation of 3278456 random numbers:

Generator state:
S = {
    6683,      115,      4964,      7660,      4600
}

sax.scs.fsu.edu: /home/npandey/ranproj>

```

Figure 3.2: The output of the TestU01 ALFG generator

The output produced by the matrix based leap ahead function and the ALFG generator implemented in *TestU01* library were exactly the same as shown above. This proves the

correctness of the algorithm.

3.3.2 Polynomial Method

Another common method for computing the j th state in the period of the ALFG is based on the polynomial algebra instead of the matrix algebra [1]. In this method x^p th element in the sequence of the ALFG can be represented by the function $r(x) = x^p \pmod{M}$. Since $x^p \pmod{M}$ is the p th element in the sequence and $x_p = x_{p-q} + x_0 \pmod{M}$ we have that $x^p = x^{p-q} + 1$ and $r(x) = x^{p-q} + 1 \pmod{M}$. It can be written as $r(x) = x^p \pmod{f(x)}$ with $f(x) = x^p - x^{p-q} - 1 \pmod{M}$. The notion $s(x) \pmod{f(x)}$ means that we replace the higher powers in $s(x)$ by the identity $f(x) = 0 \pmod{M}$ until the resulting polynomial has degree less than the degree of $f(x)$. In this case, we replace x^p by $x^{p-q} + 1 \pmod{M}$ until the resulting degree in $s(x)$ is less than p [1]. Instead of p th element, if we want to compute j th element in the ALFG's period then,

$$\begin{aligned} r(x) &= x^j \pmod{f(x)} \\ &= \sum_{i=0}^{p-1} c_i x^i \end{aligned}$$

With initial seed $s_0 = [x_{p-1}, x_{p-2}, \dots, x_0]$, x_j th element can be evaluated as:

$$x_j = \sum_{i=0}^{p-1} c_i \times x_i \tag{3.5}$$

Algorithm For Polynomial Based Leap Ahead

Input: the initial state vector $s = s_0$

Output: j th state vector s_j of ALFG

Compute $r(x) = x^j \pmod{f(x)}$ using the SQM algorithm explained in section 3.5.

Compute the polynomial modulus operation using the polynomial division algorithm.

Compute x_j th element using equation 3.5. This involves two vector multiplication and addition. One vector is the initial seed and the other is the coefficient vector returned by SQM function as shown in the equation (3.5).

Compute the rest of the element $x_{j+1}, x_{j+2}, \dots, x_{j+p-2}, x_{j+p-1}$ by the successive $p - 1$ polynomial multiplication of the $r(x)$ by $x \pmod{f(x)}$. Replace the $r(x)$ by the product of polynomial multiplication each time.

Return the x_j th vector

Algorithm 2: Polynomial based leap ahead

Efficiency of Algorithm Computation of x^j can be accomplished with $O(\log_2 j)$ polynomial - polynomial modular multiplication with the use of well-known SQM algorithm [1]. The cost of polynomial - polynomial modular multiplication with the polynomial of degree $p-1$ is p^2 modular multiplications and p^2 modular additions. This computes a polynomial of degree $2p-2$ and must be further reduced to a polynomial of degree $p-1$ by using the identity $f(x) \equiv 0 \pmod{M}$. This will require at most $p-1$ reductions with $p-1$ additions. This makes the overall cost at most p^2 modular multiplications and $p^2 - 2p + 1$ modular additions. Overall complexity of the polynomial based leap ahead is $p^2 \log_2 j$. Storage requirement for this method is $2 \times p$.

Interface for polynomial based leap ahead Implementation of polynomial based leap ahead involves definition of data structure required for representing polynomials.

```
typedef struct {
    long value;
} poly_coeff;

typedef struct {
    Long degree;
    poly_coeff coeff [MAX+1];
} polynom;
```

The *MAX* is set as $2 \times$ the largest lag possible.

Following are the interfaces for the polynomial based leap ahead function.

```
int init_seed(int *seed, int lag);
```

This function initializes the seed vector. The size of a seed vector is the value of the largest lag.

```
int setup_divisor(polynom *p, int lag, int k);
```

This function setup the divisor. Divisor is a polynomial $p(x) = x^{\text{lag}} - x^{\text{lag}-k} - 1$. This function initializes the polynomial p with coefficient for x^{lag} , $x^{\text{lag}-k}$ and x^0 .

```
int Sqm(Long leap, int lag, Long mod, polynom *q, polynom *r);
```

This function computes $x^j \pmod{x^p - x^{p-q} - 1 \pmod{M}}$ by using the SQM algorithm. This involves the modular polynomial multiplication and division operations. Variables *leap* is a leap ahead number (j), *mod* is a modulus (M), q is a polynomial $(x^p - x^{p-q} - 1)$ and r is a polynomial returned by the function as a result of the SQM.

```
int mult_polynom(polynom *p, polynom *q, polynom *r);
```

This function multiplies the polynomial p with q and returns the product in the polynomial r .

```
int div_polynom(polynom *p, polynom *q, polynom *r);
```

This function divides the polynomial q by polynomial p and returns the remainder in the polynomial r .

```
Long * get_state(Long *s, polynom *q, polynom *r, int lag,  
                unsigned long M);
```

This function returns an array of lag number of values after the x^j th element. Each of the $x_{j+1}, x_{j+2}, \dots, x_{j+lag-1}$ elements is computed by the modular multiplication of polynomial $r(x)$ by the polynomial x^1 where the polynomial $r(x)$ is represented by the variable r .

```
Long *run_poly(Long j, int l, int k, Long mod, polynom *Q, polynom  
              *R);
```

This is a wrapper function to call the polynomial leap ahead functions.

Correctness of The Algorithm The implementation of the polynomial based leap ahead algorithm was tested in two ways. First method is called as the *unit testing*. Using the concept of *unit testing*, each function was tested independently of the other functions. For instance, the major component of this algorithm was SQM, polynomial division and multiplication. These functions were tested alone with various inputs before they are integrated with other functions.

Second method was used to test the overall output of the algorithm with some standard ALFG generator. We used the *TestU01* library's ALFG generator to compare the state of the generator given by the output of the polynomial based leap ahead algorithm. The output of the polynomial based leap ahead algorithm after leaping n numbers ahead was compared with the state of the *TestU01* ALFG generator after generating n numbers of random numbers. We found the identical output for extensive number of inputs. Here is an instance of the various tests conducted to verify the correctness of the algorithm. The parameters for the ALFG generator as defined in the equation (2.4) were $p = 5$, $k = 3$, $M = 2^{13}$ and the leap ahead number as defined in the equation (3.4) was $j = 3278456$. Following is a snippet of code to call polynomial leap ahead implementation.

```

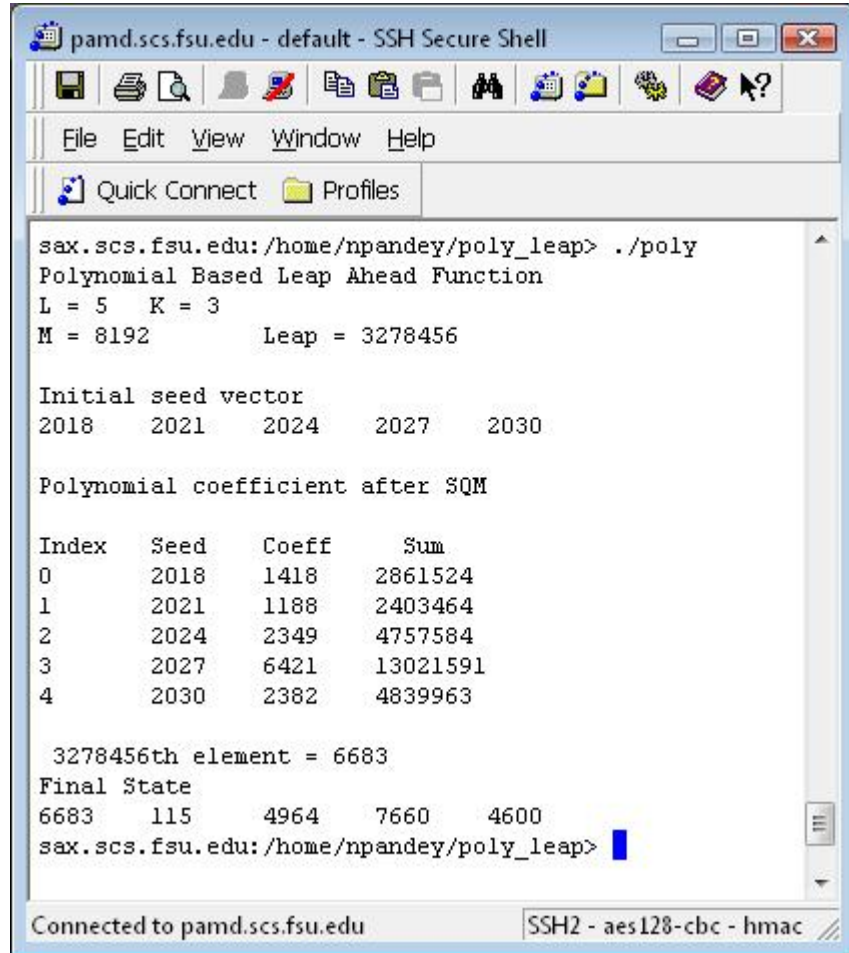
#include "polynoms.h"

polynom *Q, *R;
int main() {
    Long *S;
    int i, l, k;
    Long mod, j;

    //dynamic memory allocation of polynomials
    Q = (polynom *) calloc(1, sizeof(polynom));
    R = (polynom *) calloc(1, sizeof(polynom));
    l = 5; k = 3; j = 3278456; mod = 13;
    S = run_poly(j, l, k, mod, Q, R);
    printf("\nFinal State\n");
    for(i = 0; i < l; i++)
        printf("%u\t", S[i]);
    printf("\n");
    free(S);
    return 0;
}

```

Here is the screenshot of the output of the above program.



```
sax.scs.fsu.edu: /home/npandey/poly_leap> ./poly
Polynomial Based Leap Ahead Function
L = 5   K = 3
M = 8192      Leap = 3278456

Initial seed vector
2018   2021   2024   2027   2030

Polynomial coefficient after SQM

Index   Seed   Coeff   Sum
0       2018   1418   2861524
1       2021   1188   2403464
2       2024   2349   4757584
3       2027   6421   13021591
4       2030   2382   4839963

3278456th element = 6683
Final State
6683   115   4964   7660   4600
sax.scs.fsu.edu: /home/npandey/poly_leap>
```

Figure 3.3: The output of the polynomial based ALFG algorithm

Similarly, refer the code and the figure 3.3.1 to run the ALFG generator implemented in the *TestU01* library. Note that *TestU01* library should be installed before running this code. For further instructions on how to install and use *TestU01* library, refer the document [6].

The output produced by the polynomial based leap ahead function and the ALFG generator implemented in *TestU01* library were exactly the same. This proves the correctness of the algorithm.

3.4 Leap ahead for Multiplicative Lagged Fibonacci Generator(MLFG)

The recurrence relation for the MLFG is given by the equation (2.5). It's maximal period is $P = 2^{m-3}(2^p - 1)$ [5]. The MLFG can be parallelized based on the observation of Marsaglia. Since the product of an even number and another number is an even number, if the seed of a MLFG has a single even number in it, the MLFG sequence eventually becomes all even. However, if all the integers in the MLFG seed start out odd, then they remain odd. It eliminates the approach of being uniformly add as there is an approach of being uniformly even. Therefore, it is necessary to seed MLFGs with only odd integers. Given that only odd numbers are used as seed, we know that any odd integer x modulo 2^b can be uniquely represented as $x = (-1)^y 3^z \pmod{2^b}$, where $y \in 0, 1$ and $z \in 0, 1, \dots, 2^{b-2} - 1$ [5]. Substituting these relationships into equation (2.5), we get the following equations for the MLFG based on ALFG:

$$x_n = (-1)^{y_n} 3^{z_n} \pmod{2^b}, \quad (3.6)$$

where y_n is given by the recurrence

$$y_n = y_{n-q} + y_{n-p} \pmod{2}, \quad (3.7)$$

and z by the recurrence

$$z_n = z_{n-q} + z_{n-p} \pmod{2^{b-2}}. \quad (3.8)$$

Equations (3.7) and (3.8) are recurrences for ALFG with periods $2^p - 1$ and $2^{b-3}(2^l - 1)$ respectively. If p initial values for the y_n and z_n are given such that z_n sequence has its maximal period then the corresponding p initial values for the sequence x_n can be obtained as shown in equation (3.6).

Parallelization This project's implementation of MLFG leap ahead function follows Makino's parallelization scheme [9]. Makino parallelizes via cycle division and blocking by choosing the seeds so that the i th random number stream will produce the sequence x_{Bi}, x_{Bi+1}, \dots , where B is the block-size. It is not feasible for the processor i to generate the entire subsequence $x_0, \dots, x_{Bi}, \dots, x_{Bi+p-1}$ to start its sequence; therefore a leap ahead facility

is required provided that it enables fast computation of $x_{Bi}, \dots, x_{Bi+p-1}$, given x_0, \dots, x_{p-1} . Makino's scheme consists of, equivalently, choosing seed y_0, \dots, y_{p-1} and maximal period seed z_0, \dots, z_{p-1} . Once the state values of the y_n and z_n sequences from each stream are computed, the value for the x_n sequences can be obtained by using the equation (3.6).

Algorithm for the matrix/polynomial based Leap ahead function of the MLFG

Input: the state $y = y_0$, the state $z = z_0$

Output: j th state x_j of MLFG

Compute y_j th element using algorithm 1 for matrix based leap ahead and algorithm 2 for polynomial based leap ahead

Compute z_j th element using algorithm 1 for matrix based leap ahead and algorithm 2 for polynomial based leap ahead

Compute x_j using equation (3.8)

Return x_j

Algorithm 3: Leap ahead for MLFG

Implementation of the leap ahead function for the MLFG The equations (3.6), (3.7) and (3.8) shows that the MLFG generator is a combination of the two ALFG generators. The implementation of the MLFG leap ahead function can be done by reusing the ALFG's leap ahead function's sub-routines. Once the values of y_n and z_n of the equation (3.6) are obtained, these values are combined as shown in the equation (3.6) to get the values for x_n . The implementation of MLFG leap ahead function involves the implementation of seed initialization function for the equation (3.7) and (3.8). The leap ahead values for each of these equations (3.7) and (3.8) are obtained by using the leap ahead function that is implemented for the ALFG. Since taking a MLFG seed and finding the corresponding ALFG seed requires solving the discrete logarithm problem, we left this part as a future work. Due to this we couldn't verify the correctness of the algorithm.

3.5 Square and Multiplication Algorithm

Square and multiply algorithm is used in all of the leap ahead functions that are implemented in this project. So, is is felt necessary to list the algorithm for the better understanding of its working. The square and multiply algorithm for computing $z = a^j \pmod{M}$ with n bit integer is as follows:

```

z = a;
for( i = n - 1; i >= 0; i--){

```

```
    if(j >> i == 0)
        z = z * z ( mod M);
    else
        z = z * a ( mod M);
}
```

CHAPTER 4

Parallel Random Number Testing

Leap ahead functions for PRNGs can be used to split the stream of random numbers into various substreams. This chapter describes the test routines provided in *TestU01* and its usage to test the LCG, ALFG and MLFG parallelly using leap ahead functions described in chapter 3.

4.1 PRNG Tests

Random numbers are used in various applications ranging from simulations to sampling procedures. The difficulty associated with using true random numbers generated by the combination of hardware and software led to the development of PRNGs, which are basically a computer program that generates a sequence of random numbers. In the course of this development, testing procedures were designed to ensure that the necessary deterministic sequence of numbers produced by these PRNGs had analytical and statistical properties which compared well with those of a true random stream.

Before concluding a particular PRNG is a good one, one needs to fully test the generators against various RNT available to make sure that PRNG has all the necessary properties that satisfies the requirement of an application. This test generally involves the consumption of huge number of random numbers. Testing billions of random number against number of RNTs takes a lot of computation time. If we could test these PRNGs in parallel then computation time can be greatly reduced. For parallel RNT, PRNGs should have the property of leap ahead. Normally, a leap ahead function can be easily implemented for PRNGs based on linear recurrence. For non-linear recurrences it becomes harder to get leap ahead function that are fast and efficient in-terms of memory. In this project we have implemented fast leap ahead function for LCG, ALFG and MLFG as described in chapter

3. We will use these leap ahead functions to test LCG, ALFG and MLFG in parallel using the RNT routines implemented in *TestU01*. *TestU01* has predefined batteries of tests that included standard statistical test, with fixed parameters. List of different types of tests that are included in the batteries of tests available in *TestU01* are listed in the reference [6]. Following are the predefined batteries of tests that are available in *TestU01*.

1. *SmallCrush* It is a small and fast battery of RNTs. The function `bbattery_SmallCrush(unif01_Gen *gen)` calls the smallerush for a particular generator `gen`. This battery of test applies each tests on one unbroken stream of successive numbers. There are 10 tests defined in smallerush [6].
2. *Crush* It is a suite of stringent statistical tests. The function `bbattery_Crush(unif01_Gen *gen)` calls a collection of 96 RNTs that includes the classical tests described in Knuth [2] as well as many other tests. Most of these tests are described in chapter 2. On a PC with an AMD Athlon 64 Processor 4000+ of clock speed 2400 MHz running with Red Hat Linux, Crush will require around 1 hour of CPU time. Crush uses approximately 2^{35} random numbers. Crush uses 96 RNT out of 31 different tests. [6]
3. *Big Crush* It is a suite of very stringent statistical tests. On a PC with an AMD Athlon 64 Processor 4000+ of clock speed 2400 MHz running with Linux, BigCrush will take around 8 hours of CPU time. *BigCrush* uses close to 2^{38} random numbers. It uses 106 RNT out of 31 different tests. [6].

4.2 Design of Parallel RNT

This project implements the modified Crush routine to test PRNGs in parallel using the leap ahead functions as explained in chapter 2. It uses the Message-Passing Interface (MPI) library in C programming language to parallelize the RNT routines.

Using Message Passing Interface (MPI) to Parallelize

MPI library is used to parallelize RNT. The idea of parallelization is to run individual tests on each node of a cluster. The total number of nodes on a cluster and the current node on which a process is running can be known by using the following MPI functions.

```
MPI_Comm_size(MPLCOMM_WORLD, &size);
```

Total number of nodes in a cluster that the user is interested to use is returned into *size* variable.

```
MPI_Comm_rank(MPLCOMM_WORLD, &rank);
```

Current node number is returned by *rank* variable.

These above functions are the building blocks of our parallelization scheme. Following are the project's requirement to run RNT in parallel:

1. Run each RNT in a node. This is enabled by knowing the rank of a node. Once the rank and size are known, we can cycle through all the nodes to run multiple tests in a node if required.
2. Call leap ahead function for a PRNG from each of the nodes. This is required because each node consumes the random number in its range. The range is the sub-stream of random numbers split by leap ahead function. In other word, each node gets a non-overlapping range of random numbers from the period of the PRNGs. Let's say if B is the block of random number to be tested by each node then $x_0, x_B, x_{2B}, \dots, x_{(n-1)B}$ are the starting state of the PRNG for the nodes $0, 1, 2, \dots, n - 1$ respectively. In other word $x_0, x_B, x_{2B}, \dots, x_{(n-1)B}$ are the number jumped ahead in the period of a PRNG by the nodes $0, 1, 2, \dots, n - 1$ respectively. This strategy is also called blocking [9] where B number of random numbers are tested by test routine running in each of the nodes.

List of *TestU01* RNTs used in modified Crush

Crush battery of RNTs available in *TestU01* is modified in-order to test leap ahead functions that are implemented in this project. Following are the list of RNT tests that are used in modified crush.

1. *Serial Over (Marsaglia)* Number of RN tested: $n \times N$
2. *Collision Over (Marsaglia)* Number of RN tested: $n \times N$
3. *Birthday Spacing (Marsaglia)* Number of RN tested: $n \times N$
4. *Sim Poker(Knuth)* Number of RN tested: $n \times k$

5. *Coupon Collector (knuth)* Number of RN tested: $\approx n \times d$
6. *Gap Test (knuth)* Number of RN tested: $\frac{n}{(\text{beta}-\text{alpha})}$
7. *Run Test (knuth)* Number of RN tested: n
8. *Permutation (knuth)* Number of RN tested: $n \times t$
9. *CollisionPermut(knuth)* Number of RN tested: $N \times n \times t$
10. *Maximum-of-t(knuth)* Number of RN tested: $N \times n \times t$
11. *Smarsa_Matrix_Rank* if $k <> s$, select $\max(k, s)$. $x = \lceil \frac{k}{s} \rceil$
Number of RN tested: rows of $\max(k, s) \times x \times n$
12. *Svaria_SampleProd* Number of RN tested: $n \times t$
13. *Svaria_SampleMean* Number of RN tested: $N \times n$
14. *Svaria_SampleCorr* Number of RN tested: $N \times n$
15. *Svaria_AppearanceSpacings* $x = L\%S; y = \frac{L}{S}$
Number of RN tested: $(Q \times y + z) + (K \times y + z)$
16. *Svaria_WeightDistrib* Number of RN tested: $N \times n \times k$
17. *Svaria_SumCollector* Number of RN tested: random
18. *Smarsa_Savir2* Number of RN tested: $N \times n \times t$

Refer Appendix A [A.2](#) for interface details of above tests included in modified crush.

4.3 Performance and Result

As the motivation for the implementation of the leap ahead function as explained in chapter [3](#) was to use these functions to test PRNGs in parallel, we ran 44 combinations of 18 different tests from *TestU01* listed in [4.2](#) on a cluster of 31 nodes using MPI to parallelize the process of RNT, where each of these 44 tests were ran on a separate node in a round robin fashion. We call this list of tests as shown in [4.2](#), [A.2](#) as modified crush, because it is a subset of *crush* [\[6\]](#) battery of RNT available in *TestU01*.

4.3.1 Experiment Design

The experiment is to run the modified crush for the LCG and ALFG both in parallel and sequential and compare the performance results with their sequential counterparts. The correctness of each leap ahead function is already shown in chapter 3. This experiment is designed to show that these leap ahead functions can be used to parallelize RNTs.

For LCG, we use linear congruential generator implemented in *TestU01* under *ulcg* module with parameters $a = 397204094$, $m = 2147483647$, $c = 0$ and $s = 12345$. Following is the interface for LCG implemented in *TestU01* [6].

```
unif01_Gen * ulcg_CreateLCG (long m, long a, long c, long s);
```

For ALFG, we use the standard implementation of ALFG available in *TestU01* with $l = 17$, $k = 5$ and seeds were taken ranomly. Following is the interface for ALFG implemented in *TestU01* [6].

```
unif01_Gen * umrg_CreateLagFibFloat (int k, int r,  
char Op, int Lux, unsigned long S []);
```

4.3.2 Result

The experiment was done on an AMD Opteron Core Duo Processor (2.0 GHz) with 2 Gbytes of Memory. Each experiment was done 3 times and its average value was taken. Following is the result of the experiment.

<i>Generator</i>	<i>LeapAheadFunction</i>	<i>Sequential</i>	<i>Parallel</i>
		<i>CPU(sec)</i>	<i>CPU(sec)</i>
LCG	LCG	35.09	6.86
ALFG	Polynomial	36.28	9.48
ALFG	Matrix	35.84	9.62

CHAPTER 5

Concluding Remarks

5.1 Conclusion

Leap ahead functions for the LCG and the lagged Fibonacci generators have been implemented in this project. Leap ahead function for the LCG is an implementation of a relatively simple recurrence relation as shown in equation (3.2). Implementation of this function mainly involves computation of a^j , where j is the leap ahead amount. Using the SQM algorithm as shown in section 3.5, a^j can be computed in $\log_2 j$ modular multiplication. Implementation of the leap ahead function for the ALFG is more complicated than LCG's. The matrix and the polynomial based leap ahead functions are implemented for the ALFG. Since matrix based implementation uses $3 \times p^2$ memory space, it is infeasible in a situation where the lag values are larger like $p = 1279$. The polynomial based implementation is handy in such a situation, as its memory requirement is $2 \times p + 1$. The matrix based leap ahead implementation generates the j th state by using $O(p^2 \log_2 j)$ number of operations, whereas the polynomial based implementation generates the j th state by using $O(p \log_2 j)$ number of operations.

The motivation for the implementation of the leap ahead function was to use these functions to test PRNGs. Therefore, another part of this project was to use these leap ahead functions to test PRNGs in parallel. To use any leap ahead function to test PRNGs, the PRNG should allow RNT routines to set its internal state so that the RNT routine can generate random numbers after certain amount of jump ahead in the period of the PRNG. This project tests LCG, ALFG and MLFG implemented in *TestU01* in parallel using the leap ahead function implemented in this project. The RNT routines are used from the *TestU01* itself. *Crush* battery of tests implemented in *TestU01* is modified to have 44

different combinations of 18 different tests to fit our requirements. We found parallel version of modified crush explained in appendix A.2 is around 3.5 times faster than the sequential version. This result was obtained while running the modified crush for LCG and ALFG using the leap ahead functions implemented in this project on a cluster of 31 nodes, where each of the 44 RNTs were given to each node on a cluster in a round robin fashion.

5.2 Future Work

The purpose of this project was to implement the leap ahead function for LCG, ALFG and MLFG and use them to test respective PRNGs in parallel. So far we have implemented and proved the correctness of the implementation as described in chapter 3. Most of the PRNGs that we came across during the implementation of this project don't have any interface to change their internal state (seed values). So, the obvious future work in this project is to further explore and implement routines that use these leap ahead functions to test various sequential PRNGs available in the market. MLFG leap ahead function uses the property of ALFG to leap ahead. But taking a MLFG seed and finding the corresponding ALFG seed requires solving the discrete logarithm problem. It is left as the future work. Finally, during the conception of this project, we planned to implement the leap ahead functions and use them to parallelize *crush* and *big crush* batteries of tests available in *TestU01*. However, due to time limitation, we have implemented a modified subset version of the *crush* as explained in chapter 4. Further extension of the current implementation to include all the RNTs available in *crush* and *big crush* batteries of tests is left as the future work.

APPENDIX A

Appendix A

A.1 Introduction to TestU01

TestU01 is a software library implemented in the ANSI C language. It provides different utilities for the empirical statistical testing of PRNGs. It provides general implementation for the most of the classical statistical tests for PRNGs along with other proposed in the literature. Three predefined batteries of tests namely *small crush*, *crush* and *big crush* for the sequence of random numbers over the interval (0,1) and for bit sequences are available. Various tools to perform systematic studies of the interaction between a specific test and the structure of the point sets produced by a given family of PRNGs are also offered. For e.g. given a kind of test and class of PRNGs, there are tools implemented in *TestU01*, that can be used to determine how large should be the sample size of the test, as a function of the generator's period length, before the generator starts to fail the test systematically. Besides these, the library also provides the generic implementation for various types of PRNGs along with specific PRNGs that are found in widely used software or read literature [6]. Various statistical tests available in *TestU01* can be applied to the generators already implemented in the library or to the user-defined generators.

Organization of TestU01 library The software tools of *TestU01* are organized in four classes of modules: those implementing RNGs, those implementing statistical tests, those implementing pre-defined batteries of tests, and those implementing tools for applying tests to entire families of generators. The names of the modules in those four classes start with the letters u, s, b, and f, respectively. The name of every public identifier is prefixed by the name of the module to which it belongs. Further details on *TestU01* can be found in the reference [6].

Uniform Generator The module *unif01* contains the basic utilities for defining, manipulating, filtering, combining, and timing generators. Each generator must be implemented as an object of type *unif01_Gen*.

```
typedef struct {
    void *state;
    void *param;
    char *name;
    double (*GetU01) (void *param, void *state);
    unsigned long (*GetBits) (void *param, void *state);
    void (*Write) (void *state);
} unif01_Gen;
```

The function *GetU01* returns a floating-point number in $[0, 1)$ while *GetBits* returns a block of 32 bits. The function *Write* will write the current state of the generator. The string *name* describes the current generator, its parameters, and its initial state.

Here is the list of interface for the PRNGs LCG, ALFG and MLFG that are implemented in *TestU01*.

- *Linear Congruential Generator (LCG)*

```
unif01_Gen * ulcg_CreateLCG (long m, long a, long c, long s);
```

The initial state is $x_0 = s$ and the output at step i is x_i/m . The actual implementation depends on the values of (m, a, c) . Restrictions: a, c and s must be non-negative and less than m .

- *Lagged Fibonacci Generator (ALFG)*

```
unif01_Gen * umrg_CreateLagFib (int t, int k, int r,
    char Op, int Lux, unsigned long S[]);
```

The parameter *Op* may take one of the values $*$, $+$, $-$, x , which stands for multiplication, addition, subtraction, and exclusive-or respectively.

A.2 PRNG tests in modified crush

1. *Serial Over (Marsaglia)* Number of RN tested: $n \times N$

```
void smarsa_SerialOver (unif01_Gen *gen, sres_Basic *res,
    long N, long n, int r, long d, int t);
```


Implements the overlapping t-tuple test. It is similar to `sknuth_Serial` described in [2], except that the n vectors are generated with overlap, as follows. A sequence of uniforms u_0, \dots, u_{n-1} is generated, and the n points are defined as $(u_0, \dots, u_{t-1}), (u_1, \dots, u_t), \dots, (u_{n-1}, u_n, u_0, \dots, u_{t-3}), (u_n, u_0, \dots, u_{t-2})$.

2. *Collision Over (Marsaglia)* Number of RN tested: $n \times N$

```
void smarsa_CollisionOver (unif01_Gen *gen, smarsa_Res *res,
long N, long n, int r, long d, int t);
```

Similar to the collision test, except that the vectors are generated with overlap, exactly as in `smarsa_SerialOver`. This test corresponds to the test overlapping pairs sparse occupancy (OPSO) test described in [6] and studied by Marsaglia and Zaman. Let $\lambda = (n - t + 1)/d^t$, called the density. If n (the number of points) and d^t (the number of cells) are very large and have the same order of magnitude then, the number of collisions C is a random variable which is approximately normally distributed with mean $\mu \approx d^t(\lambda - 1 + e^\lambda)$ and variance $\sigma^2 \approx d^t e^{-\lambda}(1 - 3e^{-\lambda})$.

3. *Birthday Spacing (Marsaglia)* Number of RN tested: $n \times N$

```
void smarsa_BirthdaySpacings (unif01_Gen *gen, sres_Poisson *
    res,
long N, long n, int r, long d, int t, int p);
```

This is a variation of the collision test, in which n points are thrown into $k = d^t$ cells in t dimensions as in `smultin_Multinomial`. The cells are numbered from 0 to $k - 1$. To generate a point, t integers y_0, \dots, y_{t-1} in $0, \dots, d - 1$ are generated from t successive uniforms. The parameter p decides in which order these t integers are used to determine the cell number: The cell number is $c = y_0 d^{t-1} + \dots + y_{t-2} d + y_{t-1}$ if $p = 1$ and $c = y_{t-1} d^{t-1} + \dots + y_1 d + y_0$ if $p = 2$.

4. *Sim Poker(Knuth)* Number of RN tested: $n \times k$

```
void sknuth_SimpPoker (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, int d, int k);
```

Applies the simplified poker test described by Knuth [2]. It generates n groups of k integers from 0 to $d - 1$, by making nk calls to the generator, and for each group it computes the number s of distinct integers in the group. It then applies a chi-square

test to compare the expected and observed number of observations for the different values of s . Restrictions: $d < 128$ and $k < 128$.

5. *Coupon Collector (knuth)* Number of RN tested: $\approx n \times d$

```
void sknuth_CouponCollector (unif01_Gen *gen, sres_Chi2 *res ,  
long N, long n, int r, int d);
```

Applies the coupon collector test described in [2]. The test generates a sequence of random integers in $0, \dots, d - 1$, and counts how many must be generated before each of the d possible values appears at least once. This is repeated n times. The test counts how many times exactly s integers were needed, for each s , and compares these counts with the expected values via a chi-square test. Restriction: $1 < d < 62$. If d is too large for a given n , there will be only 1 class for the chi-square and the test will not be done.

6. *Gap Test (knuth)* Number of RN tested: $\frac{n}{(\beta - \alpha)}$

```
void sknuth_Gap (unif01_Gen *gen, sres_Chi2 *res ,  
long N, long n, int r, double Alpha, double Beta);
```

Applies the gap test described by Knuth [2]. Let $\alpha = Alpha$, $\beta = Beta$, and $p = \beta - \alpha$. The test generates n values in $[0, 1)$ and, for $s = 0, 1, 2, \dots$, counts the number of times that a sequence of exactly s successive values fall outside the interval $[\alpha, \beta]$ (this is the number of gaps of length s between visits to $[\alpha, \beta]$). It then applies a chi-square test to compare the expected and observed number of observations for the different values of s . Restrictions: $0 \leq \alpha < \beta \leq 1$.

7. *Run Test (knuth)* Number of RN tested: n

```
void sknuth_Run (unif01_Gen *gen, sres_Chi2 *res ,  
long N, long n, int r, boolean Up);
```

Applies the test of increasing or decreasing subsequences (runs up or runs down) [2]. It measures the lengths of subsequences of successive values in $[0, 1)$ that are generated in increasing (or decreasing) order. If $Up = TRUE$, it considers runs up, otherwise it considers runs down. These subsequences are the runs. The test thus generates n random numbers, counts how many runs of each length there are after merging all run

lengths larger or equal to 6, and computes the statistic V . For large n , this V should follow approximately the chi-square distribution with 6 degrees of freedom.

8. *Permutation (knuth)* Number of RN tested: $n \times t$

```
void sknuth_Permutation (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, int t);
```

Applies the permutation test [2]. It generates n non-overlapping vectors of t values, each vector using t successive values obtained from the generator, and determines to which permutation each vector corresponds (the permutation that would place the values in increasing order). The test counts the number of times each permutation has appeared and compares these counts with the expected values ($n/t!$) via a chi-square test.

9. *CollisionPermut(knuth)* Number of RN tested: $N \times n \times t$

```
void sknuth_CollisionPermut (unif01_Gen *gen, sknuth_Res2 *res,
long N, long n, int r, int t);
```

Similar to `sknuth_Collisions`, except that instead of generating vectors as in `sknuth_Serial`, it generates permutations as in `sknuth_Permutation`. It then computes the number of collisions between these permutations. Restriction $2 \leq t \leq 18$ and $t!/n < 2^{31}$.

10. *Maximum-of-t(knuth)* Number of RN tested: $N \times n \times t$

```
void sknuth_MaxOft (unif01_Gen *gen, sknuth_Res1 *res,
long N, long n, int r, int d, int t);
```

Applies the maximum-of-t test [2]. This test generates n groups of t values in $[0, 1)$, computes the maximum X for each group, and then compares the empirical distribution function of these n values of X with the theoretical distribution function of the maximum, $F(x) = xt$, via a chi-square test and an Anderson-Darling (AD) test. To apply the chi-square test, the values of X are partitioned into d categories in a way that the expected number in each category. For $N > 1$, the empirical distribution of the p-values of the AD test is compared with the AD distribution.

11. *Smarsa_Matrix_Rank* if $k \ll s$, select $\max(k, s)$. $x = \lceil \frac{k}{s} \rceil$

Number of RN tested: rows of $\max(k, s) \times x \times n$

```
void smarsa_MatrixRank (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, int s, int L, int k);
```

Applies the test based on the rank of a random binary matrix. It fills a $L \times k$ matrix with random bits as follows. A sequence of uniforms are generated and s bits are taken from each. The matrix is filled one row at a time, using $\lceil k/s \rceil$ uniforms per row. The test then computes the rank of the matrix (the number of linearly independent rows). It thus generates n matrices and counts how many there are of each rank. Finally it compares this empirical distribution with the theoretical distribution of the rank of a random matrix, via a chi-square test, after merging classes if needed (as usual).

12. *Svaria_SampleProd* Number of RN tested: $n \times t$

```
void svaria_SampleProd (unif01_Gen *gen, sres_Basic *res,
long N, long n, int r, int t);
```

This test generates t_n uniforms u_1, \dots, u_{t_n} and computes the empirical distribution of the products of n nonoverlapping successive groups of t values,

$u_{(j-1)t+1}, u_{(j-1)t+2}, \dots, u_{jt} : j = 1, \dots, n$. This distribution is compared with the theoretical distribution of the product of t independent $U(0, 1)$ random variables.

13. *Svaria_SampleMean* Number of RN tested: $N \times n$

```
void svaria_SampleMean (unif01_Gen *gen, sres_Basic *res,
long N, long n, int r);
```

This test generates n uniforms u_1, \dots, u_n and computes their average.

14. *Svaria_SampleCorr* Number of RN tested: $N \times n$

```
void svaria_SampleCorr (unif01_Gen *gen, sres_Basic *res,
long N, long n, int r, int k);
```

This test generates n uniforms u_1, \dots, u_n and computes the empirical autocorrelation of lag k .

15. *Svaria_AppearanceSpacings* $x = L\%S; y = \frac{L}{S}$

Number of RN tested: $(Q \times y + z) + (K \times y + z)$

```
void svaria_AppearanceSpacings (unif01_Gen *gen, sres_Basic *
    res,
long N, long Q, long K, int r, int s, int L);
```

The goal of this test is to measure the entropy of a sequence of random bits. The test takes the s most significant bits (after dropping the first r) from each uniform, and concatenates these s -bit blocks to construct $Q + K$ blocks of L bits. The first Q blocks are used for the initialization, and the K following blocks serve for the test proper. For each of these K blocks, the function finds the number of blocks generated since the most recent occurrence of the same block in the sequence.

16. *Svaria_WeightDistrib* Number of RN tested: $N \times n \times k$

```
void svaria_WeightDistrib (unif01_Gen *gen, sres_Chi2 *res,
    long N, long n,
    int r, long k, double alpha, double beta);
```

This test generates k uniforms, u_1, \dots, u_k , and computes the number of u_j s falling in the interval $[\alpha, \beta)$.

17. *Svaria_SumCollector* Number of RN tested: random

```
void svaria_SumCollector (unif01_Gen *gen, sres_Chi2 *res,
    long N, long n, int r, double g);
```

It generates a sequence of uniforms u_0, u_1, \dots adds them up until their sum exceeds g . It then defines $J = \min \ell \geq 0 : u_0 + \dots + u_\ell > g$. This is repeated n times, to obtain n copies of J , say J_1, \dots, J_n , whose empirical distribution is compared to the theoretical distribution by a chi-square test.

18. *Smarsa_Savir2* Number of RN tested: $N \times n \times t$

```
void smarsa_Savir2 (unif01_Gen *gen, sres_Chi2 *res,
    long N, long n, int r, long m, int t);
```

Applies a modified version of the Savir test, as proposed by Marsaglia. The test generates a random integer I_1 uniformly in $1, \dots, m$, then a random integer I_2 uniformly in $1, \dots, I_1$, then a random integer I_3 uniformly in $1, \dots, I_2$, and so on until I_t . It thus generates n values of I_t and compares their empirical distribution with the theoretical one, via a chi-square test.

Complete description of these tests can be found at [6].

A.3 Introduction to MPI

The MPI (Message Passing Interface) standard defines a software library used to turn serial applications into parallel ones that can be run on distributed memory systems. Typically these systems are clusters of servers or networks of workstations. The MPI library is used to distribute processes to the nodes on a cluster and provide a way for the processes to communicate with each other [10]. The standard was created by the MPI Forum (www.mpi-forum.org) in 1994 and is now the de facto standard for parallel programming.

MPI has an extensive list of 125 functions. Out of which using only 6 functions most of the parallel programs can be written.

```
MPI_INIT
MPI_FINALIZE
MPI_COMM_SIZE
MPI_COMM_RANK
MPI_SEND
MPI_RECV
```

One need not master all parts of MPI to use it. To use mpi functions "mpi.h" header file should be included. This header file provides the basic MPI definitions and types. MPI programs must start with

```
int MPI_Init(int *argc, char ***argv)
```

function and exit with

```
int MPI_Finalize()
```

function. MPI functions are just library routines that can be used on top of the regular programming languages like C, C++, Fortran e.t.c. Following are some of the common MPI functions.

- **int** MPI_Comm_size(MPIComm comm, **int** *size)

Determines the number of processes within a communicator.

- **int** MPI_Comm_rank(MPIComm comm, **int** *rank)

Determine the processor rank within a communicator. Rank is determined with respect to a communicator (context of the communication). *MPI_COMM_WORLD* is a predefined communicator that includes all processes (already mapped to processors).

- **int** MPI_Send (**void** *buf, **int** count, MPI_Datatype datatype, **int** dest, **int** tag, MPI_Comm comm)

This function is used to send a message. The message buffer is described by (*buf*, *count*, *datatype*). The target process is specified by *dest*, which is the rank of the target process in the communicator *comm*. When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

- **int** MPI_Recv (**void** *buf, **int** count, MPI_Datatype datatype, **int** source, **int** tag, MPI_Comm comm, MPI_Status * status)

This function is used to receive a message. Waits until a matching (both source and tag) message is received from the system, and the buffer can be used. *source* is rank in communicator specified by *comm*, or *MPI_ANY_SOURCE* (a message from anyone) *tag* is a tag to be matched on or *MPI_ANY_TAG*. Receiving fewer than *count* occurrences of datatype is ok, but receiving more is an error (result undefined). *status* contains further information (e.g. size of message, rank of the source).

Compiling, Linking and Running MPI programs in C To compile a program

mpicc < *cprogramfile* >

To run a program

mpirun - *machinefile* < *hostmap* > -*np* < *n* > < *compiledobjectcode* >

hostmap specifies the mapping.

APPENDIX B

Appendix B

B.1 LCG Leap Ahead

lcg_leap.h

```
#ifndef _LCG_LEAP_
#define _LCG_LEAP_

#include<stdio.h>
typedef unsigned long Long;

Long mul(Long base, Long exp, Long mod);
Long lcg_leap(Long x0, Long a, Long c, Long m, Long n);

#endif
```


lcg_leap.c

```
#include "lcg_leap.h"

Long mod_mul(Long base, Long exp, Long mod){
    Long result;
    result = base;
    result %= mod;
    while(exp){
        if(exp & 1){
            //multiply
            result = result * base;
            result %= mod;
            exp = exp - 1;
        }
        //square
        base = base * base;
        base %= mod;
        exp = exp/2;
    }
    return result;
}
```

```
Long mul(Long base, Long exp, Long mod){
    Long i, temp;
    temp = base;
    temp %= mod;
    base %= mod;
    for (i = 1; i < exp ; i++){
        base *= temp;
        base %= mod;
    }
}
```

```

    }
    return base;
}

Long lcg_leap(Long x0, Long a, Long c, Long m, Long n){
    Long temp;
    Long p, prod, val1, val2;
    if(a <= 1){
        fprintf(stderr, "invalid parameter a = %d", a);
        return 0;
    }
    //initial seed 1
    if(x0 == 1){
        //multiplicative cong. generator
        if(c == 0){
            return mod_mul(a, n, m);
        }
        //mixed cong. generator
        if(c > 0){
            p = mod_mul(a, n, m);
            val1 = p - 1;
            prod = (val1 * c)%m;
            temp = prod/(a-1);
            return ((p + temp)%m);
        }
    }
    //initial seed 0
    if(x0 == 0){
        if(c == 0)
            return 0;
        if(c > 0){
            p = mod_mul(a, n, m);

```

```

    val1 = p - 1;
    prod = (val1 * c)%m;
    temp = prod/(a-1);
    return temp % m;
}
}
//initial seed greater than 1
if(x0 > 1){
    //multiplicative cong. generator
    if(c == 0){
        p = mod_mul(a,n-1,m);
        return (p*x0)%m;
    }
    //mixed cong. generator
    if(c > 0 ){
        p = mod_mul(a,n,m);
        prod = (p * x0)%m;
        val1 = p -1;
        val2 = val1 * c;
        temp = val2/(a-1);
        return ((prod + temp)%m);
    }
}
}
}

```

APPENDIX C

Appendix C

C.1 Matrix Based ALFG Leap Ahead

plfg_leap.h

```
#ifndef PLFG_UTIL_H
#define PLFG_UTIL_H
```

```
#include<stdio.h>
#include<stdlib.h>
```

```
#define VALID_LEN 12
```

```
typedef unsigned long Long;
```

```
struct vstruct{
    int L;
    int K;
    int LSBS;
    int first;
};
```

```
/* Holds the current matrix size */
int mat_size;
```

```

const struct vstruct valid[] = {{1279,861,1,233},{17,5,1,10}, {5,2,1,1},
{31,6,1,2}, {55,24,1,11},{63,31,1,14},{127,97,1,21},{521,353,1,100},
{521,168,1,83},{607,334,1,166},{3,1,1,1},{2,1,1,1}};

/* Allocate memory according to the size of matrix */
int init_matrix(int, int, Long***,Long***, Long***);

/* returns the second smaller lag */
int get_second_lag(int);

void display_matrix(Long **);

/* Takes Jump ahead and M as argument */

void sqm(Long, Long, Long **, Long **, Long **);

/* set matrix elements to zero */
int fill_zero(Long **, int);

/* initializes initial seed */
void init_seed(int, Long, Long **);

/* matrix - vector multiplication */
void vector_mult(Long, Long *, Long **, Long **);

/* wrapper function to call all functions to leap ahead */
void run_plfg(Long, Long, int,int, Long **, Long **, Long **, Long * );

/* frees allocated memory */
int mat_rel(int, Long **, Long **, Long **, Long *);

#endif /* PLFG_UTIL_H */

```

plfg_leap.c

```
#include "plfg_util.h"

int init_matrix(int size, int k, Long ***template, Long ***ident, Long ***temp){
    int i,j;
    Long **template_matrix, **ident_matrix, **temp_matrix;
    mat_size = size;

    /* create template matrix*/
    if((template_matrix = (Long **)malloc(sizeof(Long *)*size))== NULL){
        printf("Cannot allocate memory\n");
        return 0;
    }

    for(i = 0; i<size; i++)
        *(template_matrix+i)=(Long *)malloc(sizeof(Long)*size);
    *template = template_matrix;

    /* create ident matrix */
    ident_matrix = (Long **)malloc(sizeof(Long *)*size);
    for(i = 0; i < size; i++)
        *(ident_matrix+i) = (Long *)malloc(sizeof(Long)*size);
    *ident = ident_matrix;

    /* create temp matrix */
    temp_matrix = (Long **)malloc(sizeof(Long *)*size);
    for(i = 0; i < size; i++)
        *(temp_matrix+i) = (Long *)malloc(sizeof(Long)*size);
    *temp = temp_matrix;
```

```

    /* set the values for template matrix */
    for(i = 0; i < size; i++){
        for(j = 0; j < size; j++){
            if(i == 0){
if(j == k-1 || j==size-1)
                template_matrix[i][j] = 1;
            else
                template_matrix[i][j] = 0;
            }else{
if(j == i-1)
                template_matrix[i][j] = 1;
            else
                template_matrix[i][j] = 0;
            }
        }
    }

    /* set the values for identity matrix */
    for(i = 0; i < mat_size; i++){
        for(j = 0; j < mat_size; j++){
            if(i ==j){
ident_matrix[i][j] = 1;
            }
            else{
ident_matrix[i][j] = 0;
            }
        }
    }
    return 0;
}

int fill_zero(Long **arr, int size){

```

```

int i, j;
for(i = 0; i < size; i++)
    for(j = 0; j < size; j++)
        arr[i][j] = 0;
}

```

```

int get_second_lag(int size){
int i;
for(i = 0; i < VALID_LEN; i++){
    if(size == valid[i].L)
        return valid[i].K;
}
}

```

```

void display_matrix(Long **arr){
int i,j;
for(i = 0; i < mat_size; i++){
    for(j = 0; j < mat_size; j++){
        printf("%d\t", arr[i][j]);
    }
    printf("\n");
}
}

```

```

void sqm(Long leap, Long M, Long **template_matrix, Long **ident_matrix, Long **temp_ma
int i, j, k;
int cnt;
cnt = 0;

while(leap){

```



```

    if(leap & 1){
        for(i = 0; i < mat_size ; i++){
for(j = 0; j < mat_size; j++){
    for(k = 0; k < mat_size; k++){
        temp_matrix[i][j] += ident_matrix[k][j] * template_matrix[i][k] ;
        temp_matrix[i][j] = temp_matrix[i][j] % M;
    }
}
    }
    for(i = 0; i < mat_size; i++){
        for(j = 0; j < mat_size; j++){
ident_matrix[i][j] = temp_matrix[i][j];
        }
    }
    fill_zero(temp_matrix, mat_size);
    leap--;
}

    for(i = 0; i < mat_size ; i++){
        for(j = 0; j < mat_size; j++){
for(k = 0; k < mat_size; k++){
    temp_matrix[i][j] += template_matrix[i][k] * template_matrix[k][j] ;
    temp_matrix[i][j] = temp_matrix[i][j] % M;
}
        }
    }
    for(i = 0; i < mat_size; i++){
        for(j = 0; j < mat_size; j++){
template_matrix[i][j] = temp_matrix[i][j];
        }
    }
    fill_zero(temp_matrix, mat_size);

```

```

    leap /= 2;
    cnt++;
}
}

```

```

void init_seed(int lag, Long M, Long **s){
    /* this function gets seed from sprng or other sources later on */
    int i;
    Long r;
    Long *seed;

    printf("Initial Seed\n");
    seed = (Long *) malloc(sizeof(Long)*lag);
    for(i = lag-1,r = 1; i >= 0; i--, r+=1){
        seed[i] = r % M;
        printf("%u\t", seed[i]);
    }
    printf("\n");
    *s = seed;
}

```

```

void vector_mult(Long M, Long *seed, Long **ident_matrix, Long **temp_matrix){
    int i, j, k;
    Long sum = 0;

    /* set the elements of temp matrix to zero */
    fill_zero(temp_matrix, mat_size);

    for(i = 0; i < mat_size ; i++){
        sum = 0;
        for(j = 0; j < mat_size; j++){
            sum += ident_matrix[i][j]*seed[j];

```

```

sum = sum % M;
temp_matrix[i][0] = sum;
    }

}

/* move the content into seed vector */
for(i = 0; i < mat_size; i++)
    seed[i] = temp_matrix[i][0];
}

int mat_rel(int lag, Long **template_matrix, Long **ident_matrix, Long **temp_matrix, L
    /* release template matrix */
    Long i;
    for(i = 0; i < lag; i++)
        free(template_matrix[i]);
    free(template_matrix);

    /* release ident matrix */
    for(i = 0; i < lag; i++)
        free(ident_matrix[i]);
    free(ident_matrix);

    /* free temp matrix */
    for(i = 0; i < lag; i++)
        free(temp_matrix[i]);
    free(temp_matrix);

    /* free seed vector */
    free(seed);

}

```

```

void run_plfg(Long leap, Long M, int lag, int k, Long **template_matrix, Long **ident_ma
    int i;
    printf("Matrix Based Leap Ahead Function\n");
    printf("L = %d\tK = %d\tM = %u\tLeap = %u\n", lag, k, M, leap);

    //initialize all matrices
    init_matrix(lag, k, &template_matrix, &ident_matrix, &temp_matrix);

    //display template matrix
    printf("Initial Template Matrix\n");
    display_matrix(template_matrix);

    //square and multiplies template_matrix, ident_matrix. Puts result into ident matrix
    sqm(leap, M, template_matrix, ident_matrix, temp_matrix);

    printf("Matrix after SQM\n");
    display_matrix(ident_matrix);

    init_seed(lag, M, &seed);

    //matrix-vector multiplication
    vector_mult(M, seed, ident_matrix, temp_matrix);

    //display final state
    printf("Final state after %d leap ahead\n", leap);
    for(i = lag - 1; i >= 0; i--)
        printf("%d\t", seed[i]);
    printf("\n");

    //free memory
    mat_rel(lag, template_matrix, ident_matrix, temp_matrix, seed);

```

```
}

int main(int argc, char *argv[]){
    Long M = 13 , leap_ahead = 3278456;
    int lag = 5 , k = 3;

    Long **template_matrix; //template matrix
    Long **ident_matrix;    //identity matrix
    Long **temp_matrix;     //temporary matrix
    Long *seed;             //initial seed vector

    M = 1 << M;
    run_plfg(leap_ahead, M, lag, k, template_matrix, ident_matrix, temp_matrix, seed);

    return 0;
}
```

C.2 Polynomial Based ALFG Leap Ahead

polynoms.h

```
#ifndef _POLY_H_
#define _POLY_H_

#define MAX 2559 // max degree of a polynomial
#define MAXLONG 214748364 // max long integer
#define SMALL 1e-12 // small real number
#define FALSE 0
#define TRUE 1

typedef unsigned long Long;

typedef struct {
    long value; // coefficient values
} poly_coeff;

typedef struct {
    Long degree; // degree of polynomial
    poly_coeff coeff[MAX + 1]; // coefficients of polynomial
} polynom;

int add_number(poly_coeff, poly_coeff, poly_coeff *);

int mult_number(poly_coeff, poly_coeff, poly_coeff *);

int div_number(poly_coeff, poly_coeff, poly_coeff *);

int setup_divisor(polynom *, int , int);

int setup_dividend(polynom *, Long);
```

```
void init_seed(Long *,int, Long);

Long polyleap(Long *, polynom *, int, Long, int);

Long *get_state(Long *, polynom *,polynom *, int, Long);

void Sqm(Long, int, Long, polynom *, polynom *);

int mult_polynom(polynom *, polynom *, polynom *);

int div_polynom(polynom *, polynom *, polynom *);

Long *run_poly(Long, int , int, Long, polynom *, polynom *);

#endif
```

polynoms.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "polynoms.h"

int add_number(poly_coeff x, poly_coeff y, poly_coeff *z) {
    z->value = x.value + y.value;
    return TRUE;
}

int mult_number(poly_coeff x, poly_coeff y, poly_coeff *z) {
    z->value = x.value * y.value;
    return TRUE;
}

int div_number(poly_coeff x, poly_coeff y, poly_coeff *z) {
    if (y.value==0)
        return FALSE;
    z->value = x.value/y.value;
    return TRUE;
}

int setup_divisor(polynom *P, int l, int k){
    int i;
    P->degree = l;
    for(i = 0; i <= l; i++){
        if(i == 0){
            P->coeff[i].value = -1;
        }
        else if(i == l-k){

```



```

        P->coeff[i].value = -1;
    }
    else if(i == 1){
        P->coeff[i].value = 1;
    }
    else{
        P->coeff[i].value = 0;
    }
}
return true;
}

int setup_dividend(polynom *Q, Long j){
    int i;
    Q->degree = j;
    for(i = 0; i <= j; i++){
        if(i == j){
            Q->coeff[i].value = 1;
        }
        else{
            Q->coeff[i].value = 0;
        }
    }
    return true;
}

void init_seed(Long *seed, int l, Long m){
    int i,r;
    for(i = 0,r = 2018; i < l; i++,r+=3){
        seed[i] = r % m;
        printf("%u\t",seed[i]);
    }
}

```

```
}
```

```
Long polyleap(Long *S, polynom *R, int l, Long mod, int DEBUG = 0){  
    int i;  
    Long sum = 0;  
  
    if(DEBUG)  
        printf("\n\nIndex    Seed    Coeff    Sum\n");  
  
    for(i = 0; i < l; i++){  
        if(DEBUG)  
            printf("%d\t%u\t%u\t",i,S[i],R->coeff[i].value);  
        sum += S[i] * R->coeff[i].value;  
        if(DEBUG)  
            printf("%u\n",sum);  
        sum = sum % mod;  
    }  
    return sum;  
}
```

```
//mutiply R(x) by x and divide the result by Q(x)
```

```
Long *get_state(Long *S, polynom *Q, polynom *R, int l, Long mod){  
    polynom *div, *temp;  
    int i;  
    Long xj,*res;  
  
    res = (Long *) malloc(sizeof(Long)*l);  
    div = (polynom *) calloc(1,sizeof(polynom));  
    temp = (polynom *) calloc(1,sizeof(polynom));  
  
    div->degree = 1;
```

```

div->coeff[1].value = 1;

for(i = 1; i <= l; i++){
    mult_polynom(div, R, temp);

    //return the result in R
    div_polynom(temp, Q, R);

    xj = polyleap(S,R,l, mod);
    res[i] = xj;
}

free(temp);
free(div);

return res ;
}

Long *run_poly(Long j, int l, int k, Long mod, polynom *Q, polynom *R){

    Long *seed, *S, xj;

    seed = (Long *) malloc(sizeof(Long)*l);
    S = (Long *) malloc(sizeof(Long)*l);

    printf("Polynomial Based Leap Ahead Function\n");
    setup_divisor(Q, l, k);
    mod = 1 << mod;
    printf("L = %u\tK = %u\n", l,k);
    printf("M = %u\tLeap = %u\n", mod, j);

    R->coeff[0].value = 1;

```

```

Sqm(j, l, mod, Q, R);

printf("\nInitial seed vector\n");
init_seed(S,l,mod);

printf("\n\nPolynomial coefficient after SQM");

xj = polyleap(S, R, l, mod, 1);
printf("\n %uth element = %u\t",j, xj);
//get rest of the elements
seed = get_state(S,Q,R,l, mod);

seed[0] = xj;

return seed;
}

void Sqm(Long leap, int l, Long mod, polynom *Q, polynom *R){
    //Q contains divisor
    //R contains remainder
    int cnt,i;
    cnt = 0;
    polynom *result, *x, *temp;

    result =(polynom *)calloc(1,sizeof(polynom));
    x =(polynom *)calloc(1,sizeof(polynom));
    temp =(polynom *)calloc(1,sizeof(polynom));

    //set x to x^1
    x->degree = 1;
    x->coeff[1].value = 1;

```

```

//initialize result with x^0
result->degree = 1;
result->coeff[0].value = 1;

while(leap){
    if(leap & 1){
        //temp contains the result
        mult_polynom(x, result, temp);
        result->degree = temp->degree;
        //put product into result polynomial
        for (i = 0; i <= temp->degree; i++){
result->coeff[i].value = temp->coeff[i].value % mod;
        }
        //compute mod
        if(div_polynom(result,Q, R)){
//R contains result
result->degree = R->degree;
for (i=0; i <= R->degree; i++)
    result->coeff[i].value = R->coeff[i].value % mod;
        }
        leap --;
    }
    //temp contains the result
    mult_polynom(x, x, temp);
    x->degree = temp->degree;
    //put product into x polynomial
    for (i = 0; i <= temp->degree; i++)
        x->coeff[i].value = temp->coeff[i].value % mod;
    //compute mod
    if(div_polynom(x , Q , R)){
//R contains result
x->degree = R->degree;

```

```

        for (i=0; i <= R->degree; i++)
x->coeff[i].value = R->coeff[i].value % mod;
    }
    leap /= 2;
    cnt++;

}

R->degree = result->degree;
for (i = 0; i <= result->degree; i++)
    R->coeff[i] = result->coeff[i];
}

//P(X) * Q(X) = R(X)
int mult_polynom(polynom *P, polynom *Q, polynom *R) {
    int i,j, n;
    poly_coeff u;
    polynom *rem;

    rem = (polynom *) calloc(1,sizeof(polynom));
    //verify that P and Q are not void

    if (P->degree == 0 && P->coeff[0].value == 0)
        return FALSE;
    if (Q->degree == 0 && Q->coeff[0].value == 0)
        return FALSE;
    rem->degree = P->degree + Q->degree;
    if (rem->degree > MAX)
        return FALSE; // R degree is too big
    for (n = 0; n <= rem->degree; n++) {
        rem->coeff[n].value = 0;
        for (i = 0; i <= P->degree; i++) {

```

```

        j = n - i;
        if (j >= 0 && j <= Q->degree) {
            if (!mult_number(P->coeff[i], Q->coeff[j], &u))
return FALSE;
            if (!add_number(rem->coeff[n], u, &rem->coeff[n]))
return FALSE;
        }
    }
}
//copy rem in R
R->degree = rem->degree;
for (i = 0; i <= rem->degree; i++)
    R->coeff[i] = rem->coeff[i];
free(rem);
return TRUE;
}

```

```

int div_polynom(polynom *P, polynom *Q, polynom *R) {
    int i,j;
    poly_coeff u;
    polynom *quo, *rem;

    quo = (polynom *) calloc(1,sizeof(polynom));
    rem = (polynom *) calloc(1,sizeof(polynom));

    //The Q polynomial must be <> zero
    if (Q->degree == 0 && Q->coeff[0].value == 0)
        return FALSE;
    rem->degree = P->degree;

    for (i = 0; i <= P->degree; i++)
        rem->coeff[i] = P->coeff[i];
}

```

```

quo->degree = P->degree - Q->degree;
if (quo->degree < 0) {
    quo->degree = 0;
    quo->coeff[0].value = 0;
}
else {
    for (i = quo->degree; i >= 0; i--) {
        if (!div_number(rem->coeff[rem->degree], Q->coeff[Q->degree], &quo->coeff[i]))
return FALSE;
        for (j = i; j <= rem->degree; j++) {
            if (!mult_number(quo->coeff[i], Q->coeff[j-i], &u))
return FALSE;
            u.value = -u.value;
            if (!add_number(rem->coeff[j], u, &rem->coeff[j]))
return FALSE;
        }
        if (rem->degree > 0)
rem->degree--;
    }
    while (fabs(rem->coeff[rem->degree].value) < SMALL && rem->degree>0)
        rem->degree--;
}
R->degree = rem->degree;
for (i = 0; i <= rem->degree; i++)
    R->coeff[i] = rem->coeff[i];
free(quo);
free(rem);
return TRUE;
}

```


run_polynoms.c

```
//#include<iostream>
#include <stdio.h>
#include <malloc.h>
#include <math.h>

#include "polynoms.h"

//using namespace std;

polynom *Q, *R;

int main() {

    Long *S;
    int i, l, k;
    Long mod,j;

    //dynamic memory allocation of polynomials
    Q = (polynom *) calloc(1,sizeof(polynom));
    R = (polynom *) calloc(1,sizeof(polynom));

    l = 5; k = 3; j = 3278456; mod = 13;

    S = run_poly(j, l, k, mod, Q,R);

    printf("\nFinal State\n");
    for(i = 0; i < l; i++)
        printf("%u\t",S[i]);
    printf("\n");
    free(S);
}
```

```
//free(P); free(Q); free(H); free(R);  
return 0;  
}
```

REFERENCES

- [1] M. Mascagni. Polynomial versus matrix methods for leap-ahead in shift-register type pseudorandom number generators. *Institute for Mathematics and its Applications (IMA) Reprint 1469*, 1997. ([document](#)), [3.1](#), [3.1](#), [3.2.1](#), [3.3](#), [1](#), [3.3.1](#), [3.3.2](#), [3.3.2](#)
- [2] D.E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Second edition*. Addison-Wesley, Massachusetts, second edition, 1981. [1](#), [2](#), [2](#), [2.5](#), [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [3.1](#), [3.2](#), [3.2](#), [3.2](#), [3.2.1](#), [2](#), [1](#), [4](#), [5](#), [6](#), [7](#), [8](#), [10](#)
- [3] D.V. Pryor M. Mascagni, M.L. Robinson and S.A. Cuccaro. Parallel pseudorandom number generation using additive lagged fibonacci recursions. 106:253–277, 1995. [2](#)
- [4] D.V. Pryor M. Mascagni, S.A. Cuccaro and M.L. Robinson. A fast high quality, and reproducible parallel lagged-fibonacci pseudorandom number generator. *Journal of Computational Physics*, 15:211–219, 1995. [2](#), [2](#)
- [5] A. Srinivasan and M. Mascagni. Parameterizing parallel multiplicative lagged-fibonacci generators. *Parallel Computing*, 34:899–916, 2004. [2](#), [1](#), [2](#), [3](#), [4](#), [3.4](#)
- [6] P. L’Ecuyer and R. Simard. Testu01 a software library in ansi c for empirical testing of random number generators user’s guide, compact version. [2.4](#), [10](#), [11](#), [12](#), [3.3.1](#), [3.3.2](#), [4.1](#), [1](#), [2](#), [3](#), [4.3](#), [4.3.1](#), [4.3.1](#), [A.1](#), [A.1](#), [2](#), [A.2](#)
- [7] M. Mascagni and A. Srinivasan. Algorithm 806: Sprng: A scalable library for pseudorandom number generation. *ACM transactions on mathematical software*, 36:436–461, 2000. [2.6](#)
- [8] P. L’Ecuyer and S. Cote. Implementing a random number package with splitting facilities. *ACM transactions on mathematical software*, 17:98–111, 1991. [3.1](#)
- [9] J. Makino. Lagged-fibonacci random number generators on parallel computers. *Parallel Computing*, 20(9):1357–1367, 1994. [3.4](#), [2](#)
- [10] Peter S. Pacheco. A user’s guide to mpi. Technical report, 1998. [A.3](#)