International Institute of
Information Technology, Hyderabad

Term Project

Techniques for Generating Random Numbers in Parallel

Final Report

Author: Kalp Shah

April 30, 2021

Contents

Chapter 1

Introduction

Randomization is defined to be the study of making something random, which can be mathematically defined as :

Definition 0.1

For a function $\mathcal{G} : \mathcal{D} \to \mathcal{R}$, if $\mathcal{G}$ is random, then :

$$P(\mathcal{G}(x) = y) = \frac{1}{\|\mathcal{R}\|}$$

The definition above says that the probability function for any input x to give an output y is uniformly random.

For a sequence of random numbers generated, two properties must hold :

- The values are uniformly distributed over a defined interval
- It is impossible to predict future values based on the past or present ones

Note 0.1

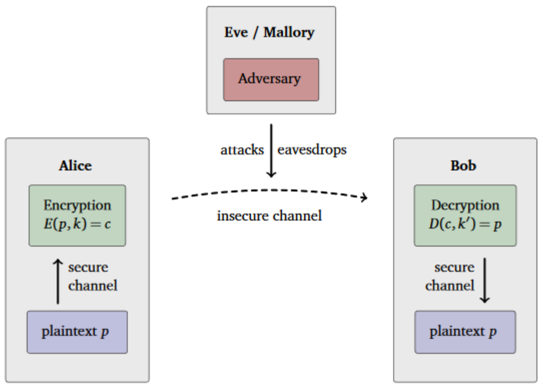
All the definitions provided here is for a function to give a random output, which is what the report will be focussing on.

1.1 Importance of Randomness

1.1.1 Cryptography

One of the most important use of random numbers is in cryptography. Cryptography is a study of techniques for secure communication in presence of third party adversary. Formulation and explaination of a cipher can easilty explain the requirement of random numbers.

A general communication channel looks as follows:



In this it can be seen that c, which is the ciphertext contains more information (higher entropy) than m but has no intrinsic meaning in itself. Therefore, it can be anything.

$E(m,k)$ is often randomized, in the case that even if the system is faulty and gets the same m and k, the ciphertext c will not be the same, thus protecting the information.

Thus, $E(m)$ is a random number generator, which is why random number generators are important in the field of cryptography.

1.1.2 Monte Carlo

Another important application of random number generation is in Monte Carlo simulation methods. Monte Carlo methods are set of algorithms that repeatedly use random sampling to obtain deterministic results.

One of the simplest and most used example of a Monte Carlo simulation problem is the determination of the value of π. The code to calculate is as below:

```
# Imports
import numpy as np

# Total number of samples to be taken
N = 100000
pCiricle = 0
pt_x = 0
pt_y = 0

for i in range(N):
        # Random values of X & Y Taken
        # X,Y in [0,1]
        pt_x = np.random.uniform()
        pt_y = np.random.uniform()

        if np.square(pt_x) + np.square(pt_y) <= 1:
            # Point in circle if x^2 + y^2 < 1
            pCiricle += 1

    # Area of Circle in First Quadrant
    # is pi/4 (r = 1), hence pi = 4*density of
    # points in circle
    pi = 4 * (pCiricle/N)
    print(pi)
```

From this, it can be seen that a deterministic value (π) is being found using random sampling of x and y points.

1.2 Pseudorandomness

Any algorithm that exists is a set of defined steps that will always give a deterministic output. Therefore, generation of true random numbers is impossible for a normal computer algorithm.

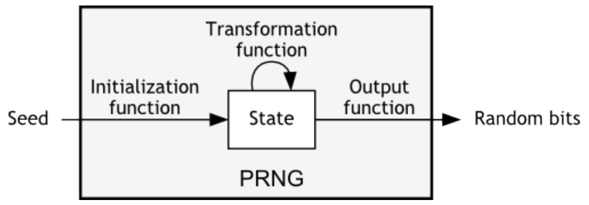
Rather if for the required amount of length, the sequence appears to be stastically random, then the job is done. An algorthm that provides with that kind of sequences are known as psuedo random number generators.

An example of one such function is :

$$\mathcal{G}(x) = g^x \mod m$$

In this, the parameters are g and m. This is a psuedo random number generator in the sense that if a predictive (non random) sequence of numbers are input as x, then the output sequence will appear to be random.

The general structure of a psuedo random number generator is as follows:



Thus, the sequence generated by a computer algorithm is not truly random, but for a particular length of output appears to be random, known as a psuedo random sequence.

Chapter 2

Sequential RNG

In this section, the most common sequential RNGs are introduced. This is required as RNGs that work in parallel are extensions of SRNGs, and understanding a simpler system before introducing a N processor generality wil be the logical and simpler way to understand PRNGs.

2.1 Linear Congruential Generators

Linear Congruential Generators, abbreviated as LCGs are defined as follows:

Definition 1.1

For a sequence of numbers $X_1, X_2, \ldots, X_n$ with n^{th} number being X_n, the sequence is :

$$X_n = aX_{n-1} + c \mod m$$

Where a, c, m are parameters.

This is one of the most common RNG, which has been used for a long time. One advantage that this construct has is that it is very quick in generation when m is chosen to be a power of 2.

The disadvantage of the system are that it produces highly correlated lower order bits for intervals that are power of 2. It also has the issue that the modulus cannot be greater than the precision fo the machine.

One modification that is commonly used for the algorithm is generalization of LCG, known as Multiple Recursive Generators (MRGs), which are defined as:

Definition 1.2

For a sequence of numbers $X_1, X_2, \ldots, X_n$ with n^{th} number being X_n, the sequence is :

$$X_n = \sum_{i=1}^{n-k} a_i X_i + c \mod m$$

Where a_i, c, m, k are parameters.

This is a recursion based some of the previous values rather than just the last one. The benefit of this construct is that the randomness of the system increases, lesser the k is, on the downside that it takes a longer time to compute.

2.1.1 Code

```
import numpy as np

def lcg(N,a,c,m,X0):
    X = [X0]
    for i in range(N):
        X.append((a*X[-1] + c)%m)

    return X

if __name__ == "__main__":
    X0 = np.random.uniform(high=100)
    c = np.random.uniform(high=100)
    a = np.random.uniform(high=100)
    m = np.random.randint(low=10000,high=99999)
    N = 10

    print(lcg(N,a,c,m,X0))
```

2.2 Lagged-Fibonacci Generators

Lagged-Fibonacci Generators, abbreviated as LFGs are defined as follows:

Definition 2.1

For a sequence of numbers $X_1, X_2, \ldots, X_n$ with n^{th} number being X_n, the sequence is :

$$X_n = X_{n-p} \odot X_{n-q} \mod m$$

Where p, q, m are parameters and $\odot$ is any binary operation (Add, Multiply ,XOR).

From the formulation itself it can be seen that the usage of the construct requires $p(p > q)$ initial values to be already present.

The binary operation can be selected by the user and it has been proven that multiply operation provides the most amount of randomness followed by the addition operation and lastly followed by XOR operation.

The major benifit of the construct is that it is extremely quick in computing the value. Its period can also be made arbitrarily wide by just changing the value of p.

2.2.1 Code

```
import numpy as np

def lfg(N,seed,p,q,m):
    seed = [int(i) for i in list(seed)]
    if len(seed) <= max(p,q):
        raise ValueError("Bigger seed required")

    X = []

    for i in range(N):
        val = (seed[p]*seed[q])%m
        seed.append(val)
        X.append(val)
        p = p + 1
        q = q + 1

    return X

if __name__ == "__main__":
    seed = str(np.random.randint(low=10000000,high
        =99999999))
    p = 3
    q = 7
    m = np.random.randint(low=10000,high=99999)
    N = 10

    print(lfg(N,seed,p,q,m))
```

2.3 Combined Generators

This is a system which just combines two psuedo random sequences, which in turn gives a random sequence which would have a better quality of randomness. Hence, the formulation is as follows :

Definition 3.1

For two sequences X_n and Y_n, making a combined sequence S_n as follows:

$$S_n = X_n + Y_n \mod m$$

Where m is the parameter.

2.4 Conclusion

It is visible that psuedo random techniques used are not that complicated, but rather tricky. The combination of the use of recursion and modulo provides us with psuedo random numbers.

The improvement in quality can be performed by performing binary operations on these sequences, which if having different parameters would have different patterns thus obscuring the pattern even more, making the operated sequence even more random.

Chapter 3

Parallel RNG

Parallel algorithms for RNGs is just a generalization or an extension of the sequential RNGs. The main technique used for such generalization is splitting.

3.1 Splitting Techniques

These algorithms work on the principle that the elements of sequential RNG must be distributed in some way among all the processors.

These systems create a single stream of random numbers much more quickly than a single processor system, but as the underlying formula for generation of random numbers remain the same, the quality of random numbers is not expected to increase.

3.1.1 Sequence Splitting

This is the simplest, where N different processors are given the task to generate P random numbers, each processor giving disjoint sequences.

The only possible issue with that even though the numbers generated are disjoint, they might still have some correlation and merging the sequence might make the quality of randomness decrease.

Code

```
import numpy as np
import importlib
import ray
from lcg import lcg
from lfg import lfg

ray.init()

ray.remote
```

```
def seqSplitLcg(N):
    X0 = np.random.uniform(high=100)
    c = np.random.uniform(high=100)
    a = np.random.uniform(high=100)
    m = np.random.randint(low=10000,high=99999)

    return lcg(N,a,c,m,X0)

proc = 4
NPerProc = 10
resultIds = []
for i in range(proc):
    resultIds.append(seqSplitLcg.remote(NPerProc))

result = []
RN = ray.get(resultIds)
for i in RN:
    result.extend(i)

print(result)
```

LaTeXis not allowing @ in the line ray.remote, which should be present. Working code provided in Code folder.

3.1.2 Random Tree Method

In the random tree method, two LCGs are used in congruence to create random sequences defined as :

$$L_{k+1} = a_L L_k \mod m$$
$$R_{k+1} = a_R R_k \mod m$$

In this, the left ones are used to generate new sequences whereas the right ones are used to generate the random numbers.

The benefits of the method are that new random sequences are produced in a reproducible and noncentralized fashion. An example of such usefulness is when dynamic amount of child process require a start number to create a random sequence, so the parent can just pass on the L_k as the parameter, while the child uses R_k for computation.

The major downside is that the quality of randomness has not increased, but rather now unknown correlations between different R_k sequences can exist.

Code

```
import numpy as np
import importlib
import ray
from lcg import lcg
from lfg import lfg

ray.init()
```

```
def LCGSplit(N,aL):
    X0 = np.random.uniform(high=100)
    c = np.random.uniform(high=100)
    m = np.random.randint(low=10000,high=99999)

    return lcg(N,aL,c,m,X0)

ray.remote
def seqSplitLcg(N,X0,aR):
    if X0 == None:
        X0 = np.random.uniform(high=100)
    c = np.random.uniform(high=100)
    m = np.random.randint(low=10000,high=99999)

    return lcg(N,aR,c,m,X0)

aL = np.random.uniform(high=100)
aR = np.random.uniform(high=100)

proc = 4
Lk = LCGSplit(proc,aL)

NPerProc = 10
resultIds = []
for i in range(proc):
    resultIds.append(seqSplitLcg.remote(NPerProc,Lk[
        i],aR))

Rk = ray.get(resultIds)

for i in range(len(Rk)):
    print(i,":",Rk[i])
```

3.1.3 Leapfrog Method

The leapfrog method is a variant of the Random Tree method specifically for systems with fixed amount of generators. In this, the constant for R_k is tweaked in the following manner :

$$L_{k+1} = a_L L_k \mod m$$
$$R_{k+1} = a_L^n R_k \mod m$$

The benefit of this method is that the sequence will not overlap for a P/n period, where P is the period of L sequence and n is the number of subsequences required. The disadvantage being that the number of subsequences required must be known.

Code

```
import numpy as np
import importlib
import ray
from lcg import lcg
from lfg import lfg
```

```
ray.init()

def LCGSplit(N,aL):
    X0 = np.random.uniform(high=100)
    c = np.random.uniform(high=100)
    m = np.random.randint(low=10000,high=99999)

    return lcg(N,aL,c,m,X0)

ray.remote
def seqSplitLcg(N,X0,aR):
    if X0 == None:
        X0 = np.random.uniform(high=100)
    c = np.random.uniform(high=100)
    m = np.random.randint(low=10000,high=99999)

    return lcg(N,aR,c,m,X0)

noSubsec = 4
aL = np.random.uniform(high=100)
aR = np.power(aL,noSubsec)

proc = 4
Lk = LCGSplit(proc,aL)

NPerProc = 10
resultIds = []
for i in range(proc):
    resultIds.append(seqSplitLcg.remote(NPerProc,Lk[
       i],aR))

Rk = ray.get(resultIds)

for i in range(len(Rk)):
    print(i,":",Rk[i])
```

3.1.4 Modified Leapfrog

One final method is known as modified leapfrog, which is a small modification on leapfrog in the case where the number of random numbers in a sequence are known but not the number of sequences. It is defined as follows :

$$L_{k+1} = a_R^n L_k \mod m$$
$$R_{k+1} = a_R R_k \mod m$$

It has the same advantages and disadvantages of the standard leapfrog method.

Code

```
import numpy as np
import importlib
import ray
from lcg import lcg
from lfg import lfg

ray.init()
```

```
def LCGSplit(N,aL):
    X0 = np.random.uniform(high=100)
    c = np.random.uniform(high=100)
    m = np.random.randint(low=10000,high=99999)

    return lcg(N,aL,c,m,X0)

ray.remote
def seqSplitLcg(N,X0,aR):
    if X0 == None:
        X0 = np.random.uniform(high=100)
    c = np.random.uniform(high=100)
    m = np.random.randint(low=10000,high=99999)

    return lcg(N,aR,c,m,X0)

noSubsec = 10
aR = np.random.uniform(high=100)
aL = np.power(aR,noSubsec)

proc = 4
Lk = LCGSplit(proc,aL)

NPerProc = 10
resultIds = []
for i in range(proc):
    resultIds.append(seqSplitLcg.remote(NPerProc,Lk[
       i],aR))

Rk = ray.get(resultIds)

for i in range(len(Rk)):
    print(i,":",Rk[i])
```

Chapter 4

Compare

Now all the algorithms will be compared in two parameters, the quality of randomness and the speed with which it is generated. The comparison algorithms are given in the appendix.

4.1 Quality of Randomness

The quality of randomness can be checked in many ways, but in this report, the randomness is checked by using the blocking test.

In this, we use the fact that sum of independant variables asymptotically approaches normal distribution. Therefore, the amount of deviation from normal distribution is the measure of randomness.

4.2 Results

Algorithm	Mean	Variance	Time
Leapfrog	11.42	9.90	0.81
Random Tree	10.86	8.90	0.88
Modified Leapfrog	11.33	10.40	0.84
Sequence Splitting	9.46	6.91	0.52

From these results it is clear that my version of these algorithms are not as precise as I would like them to be but are close enough to the normal distribution value to be considered good enough.

4.3 Video Link

The video explaining the report is uploaded on Youtube and can be opened using the following link :

https://youtu.be/j8ZJhVhkJfk

Appendix A

Testing Algorithms

In the appendix, all the codes used for testing various aspects of PRNGs is given with explanation as to why the given formulation was used.

A.1 Blocking Algorithm

```
import numpy as np
import importlib
import ray
from lcg import lcg
from lfg import lfg

ray.init()

def LCGSplit(N,aL):
    X0 = np.random.uniform(high=100)
    c = np.random.uniform(high=100)
    m = np.random.randint(low=10000,high=99999)

    return lcg(N,aL,c,m,X0)

ray.remote
def seqSplitLcg(N,X0,aR):
    if X0 == None:
        X0 = np.random.uniform(high=100)
    c = np.random.uniform(high=100)
    m = np.random.randint(low=10000,high=99999)

    return lcg(N,aR,c,m,X0)

NMax = 100000
NSeq = 1000
Xf = np.array([])
for _ in range(NSeq):
    no_subsec = 4
    aL = np.random.uniform(high=100)
    aR = np.power(aL,no_subsec)

    proc = 4
```

```
        Lk = LCGSplit(proc,aL)

        NPerProc = 10
        resultIds = []
        for i in range(proc):
            resultIds.append(seqSplitLcg.remote(NPerProc
                ,Lk[i],aR))

        Rk = ray.get(resultIds)

        val = 0
        for i in Rk:
            for j in i:
                val += j

        Xf = np.append(Xf,val)

    print("Mean :",np.mean(Xf)/NMax,"Variance :",np.var(
        Xf)/(NMax**2))
```