Parallel Monte Carlo Simulation

Ashit Rustagi, Mayank Musaddi, Tanuj Garg

April 23, 2020

This project aims to provide a detailed analysis, exploring different methods for generating random numbers in parallel and finding their application in common probabilistic algorithmic problems like Monte Carlo Simulations. Using 2 different methods for generating random numbers we have provided 4 ways to parallelize them and have successfully solved 2 math problems involving Monte Carlo Simulation. The time taken are analysed for different number of processes.

1 Overview

We have divided the report into 3 sections. The first section describes the random number generating algorithms along with their time complexities, the second section explains ways to parallelize them and the third section speaks about the implementation of the Monte Carlo techniques along with their results and analysis.

2 Random Number Generator

2.1 Introduction

A Random Number Generator (RNG) is a device that generates a sequence of numbers or symbols that cannot be reasonably predicted better than by a random chance. RNGs can be true hardware random-number generators (HRNG), which generate genuinely random numbers, or pseudo-random number generators (PRNG), which generate numbers that look random, but are actually deterministic, and can be reproduced if the state of the PRNG is known.

For the implementation of PRNG, in general there is a state Y_n, involving one or more variables, which is advanced step-by-step by a predefined algorithm.

$$Y_{n+1} = f_1(Y_n)$$

This begins from a predefined initial state Y_0. To get the n^{th} random number x_n, Y_n and a function g is used as follows,

$$x_n = g(Y_n)$$

2.2 Parallel PRNG strategies

Parallel pseudo-random number generators generate random numbers using multiple processes running parallely. This is possible if we can skip-ahead some points using some algorithm of the form

$$Y_{n+p} = f_p(Y_n)$$

This involves a small cost ($O(logp)$ in general)

We have devised 3 possible strategies for this parallelism as follows,

1. **Simple skip ahead**
   Each process skips to a particular point in the generator sequence, and then generates a contiguous segment of points. For example if there are P processes and each process generates m points, then
   Process 1 generates $x_1, x_2, ..., x_m$
   Process 2 generates $x_{m+1}, x_{m+2}, ..., x_{2m}$
   .
   .
   Process P generates $x_{(P-1)m+1}, x_{(P-1)m+2}, ..., x_{Pm}$
2. **Strided skip ahead**
   Each process generates next P^{th} point, if there are P processes.
   Process 1 generates $x_1, x_{P+1}, x_{2P+1}, ...$
   Process 2 generates $x_2, x_{P+2}, x_{2P+2}, ...$
   .
   .
   Process P generates x_P, x_{2P}, x_{3P}
3. **Hybrid**
   Combination of the above two methods

We explore 4 different parallel pseudo-random number generators techniques which using LCG and EMRG algorithms for generating random numbers and the above methods for paralellizing them.

2.3 Linear Congruence Generator

A linear congruential generator (LCG) is an algorithm that yields a sequence of pseudo-randomized numbers calculated with a discontinuous piecewise linear equation. The generator is defined by the recurrence relation given as,

$$X_{n+1} = (aX_n + b)\%m$$

Here a, b and m are pre-defined and X_0 is the seed value provided by the user.

In this algorithm if we try to jump 2 steps, the equation would look as follows,

$$X_{n+2} = (a^2X_n + (1+a)b)\%m$$

Similarly, if we jump k steps, the general equation will be as follows,

$$X_{n+k} = (a^k X_n + (1 + a + a^2 + ... + a^{k-1})b)$$

$$=> X_{n+k} = (a^k X_n + \frac{a^k - 1}{a - 1} b)\%m$$

$$=> X_{n+k} = (a^k X_n + (a^k - 1)(a - 1)^{m-2} b)\%m$$

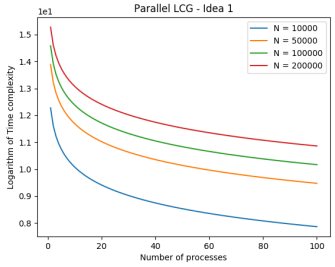
(Assuming m is prime).
Thus we could jump to the next k^{th} state in $O(log(mk))$ complexity by using Fermat's Little Theorem and Fast Modulo Multiplication Technique. Next we describe ways to parallelize this algorithm.

2.3.1 Strided Skip Ahead (LCG Idea 1)

We use the strided skip ahead strategy as explained before. Suppose there are P processes and X_i is the i^{th} random number generated, then the j^{th} process $(1 \leq j \leq P)$ generates $X_j, X_{P+j}, X_{2P+j}, ...$ random numbers.

We use following predefined values
$a = 2147483629, b = 2147483587, m = 2147483647$

Assuming that the total random numbers generated are N, thus each process generates $\frac{N}{P}$ numbers. To get to its starting random number (X_j for j^{th} process), a process takes $O(log(mP))$ time in the worst case. Now, to generate each random number, the process needs to jump by P steps. For this, it can pre-compute a^P to avoid redundant computation ($O(logP)$), but it has to compute inverse modulo for each random number it generates ($O(logm)$). Thus, time complexity of this algorithm is given by $O(log(mP) + log(P) + \frac{N}{P} log(m))$. Since no message passing is involved the message complexity is $O(1)$.

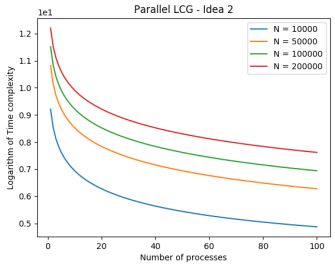


2.3.2 Simple Skip Ahead (LCG Idea 2)

We use simple skip ahead strategy as explained before. Suppose there are P processes, X_i is the i^{th} random number generated and each process generates x numbers (If N are total numbers to be generated, then $x = N/P$),then j^{th} process $(1 \leq j \leq P)$ generates $X_{(j-1)x+1}, X_{(j-1)x+2}, ..., X_{jx}$ random numbers.

We use following predefined values
$a = 2147483629, b = 2147483587, m = 2147483647$

Each process needs to reach its starting point whose worst case time complexity (for P^{th} process) is $O(log(mN)$. After reaching the starting point, a process can generate next random number in $O(1)$ and there are $\frac{N}{P}$ numbers to generate by each process. So, time complexity of this algorithm is $O(log(mN) + \frac{N}{P})$. Since no message passing is involved the message complexity is $O(1)$.



2.3.3 Hybrid Approach (LCG Idea 3)

This is an approach that uses the concepts of the above involved together. Here we use an LCG equation to generate random seeds for the parallel P processes. Then these P processes work in parallel using their own LCG's equations to generate further random numbers.
Suppose we have the following two LCG equations,

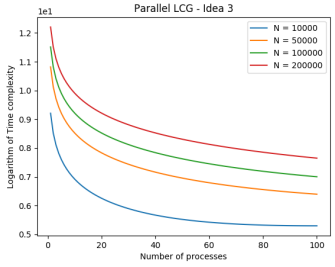
$$Y_{n+1} = (a_1 Y_n + b_1)\%m$$

$$X_{n+1} = (a_2 X_n + b_2)\%m$$

where a_1, a_2, b_1, b_2 and m are predefined.
Let there be P processes. Process 1 use the seed given as input by the user as Y_0 and generate P random numbers using the first equation. These random numbers are denoted by $Y_1, Y_2, ..., Y_P$.
Then, j^{th} process can get value of Y_k and can generate a sequence of random numbers using seed as Y_j. If we define $X_{j,i}$ to be the i^{th} random number generated by j^{th} process, then $X_{j,0} = Y_j$. Thus the j^{th} process use the second equation above to generate $\frac{N}{P}$ random numbers (Suppose N is total random numbers to be generated). Finally, they are combined and a random number sequence is obtained.

We use following prdefined values
$a_1 = 2147483629, b_1 = 2147483587, a_2 = 1140671485, b_2 = 12820163, m = 2147483647$

Time complexity of generating P numbers by a process 1 is $O(P)$. Then, time complexity to generate numbers by each process is $O(\frac{N}{P})$. Thus, total time complexity is $O(P + \frac{N}{P})$. Since a message is passed from master process to each process once, the message complexity is $O(P)$.



2.4 Ecuyer's Multiple Recursive Generator (EMRG)

This is another algorithm to generate pseudo random numbers. It has a good randomness property with a long period. It is defined by the set of equations

$$y_{1,n} = (a_{12}y_{1,n-2} + a_{13}y_{1,n-3})\%m_1,$$

$$y_{2,n} = (a_{21}y_{2,n-1} + a_{23}y_{2,n-3})\%m_2,$$

$$x_n = (y_{1,n} + y_{2,n})\%m_1,$$

for all $n >= 1$ where
$a_{12} = 1403580, a_{13} = -810728, m_1 = 2^{32} - 209,$
$a_{21} = 527612, a_{23} = -1370589, m_2 = 2^{32} - 22853.$
We initialize all required values with 1 except $y_{1,0}$ which we set to be $(seed)\%m_1$
Sequence of integers $x_1, x_2, ...$ are output random numbers of this generator.
At any point in the sequence the state can be represented by the pair of vectors

$$Y_{i,n} = \begin{pmatrix} y_{i,n} \\ y_{i,n-1} \\ y_{i,n-2} \end{pmatrix}$$

for $i = 1, 2$
Above equations can now be written as

$$Y_{i,n+1} = A_i Y_{i,n}$$

for $i = 1, 2$. Here,

$$A_1 = \begin{bmatrix} 0 & a_{12} & a_{13} \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, A_2 = \begin{bmatrix} a_{21} & 0 & a_{23} \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

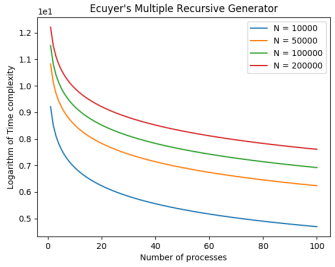
Therefore,

$$Y_{i,n+p} = A_i^p Y_{i,n}$$

for any $p >= 0$ and $i = 1, 2$.

Now, we can use simple skip ahead strategy to generate random numbers, just like in Parallel LCG Idea 2.

We can calculate A^p for a $3 * 3$ matrix A in $O(log(p))$. Therefore, worst cast time complexity to reach the start position would be for P^{th} process (P being the number of processes, N being total count of random numbers), which will be $O(log(N))$. Then, each process generates $\frac{N}{P}$ random numbers, generating each number in $O(1)$. Total time complexity is $O(log(N) + \frac{N}{P})$. Since no message passing is involved the message complexity is $O(1)$.



3 Monte Carlo methods

3.1 Introduction

Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The underlying concept is to use randomness to solve problems that might be deterministic in principle.

They tend to follow a particular pattern :

1. Define a domain of possible inputs.
2. Generate inputs randomly over the domain.
3. Perform some deterministic computation on the inputs.
4. Aggregate the results.

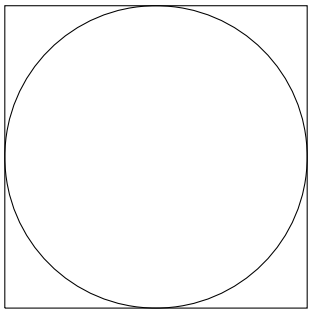
A Monte Carlo simulation can be easily implemented in parallel using multiple processes. Each process, in parallel, generates random inputs and performs some computation on them. Then one process aggregate all the results from every process and produce the required output.

Monte Carlo simulations are generally used in three problem classes: optimization, numerical integration, and generating draws from a probability distribution. In this project, we will calculate the approximate value of π and of the integral $\int_0^1 e^{-\frac{x^2}{2}} dx$.

3.2 Approximation of π

Let us take a square of side 2 units and a circle of radius 1 unit, both centered at $(0, 0)$ in cartesian plane. We know that area of the square is 4 square units and that of the circle is π square units. Thus the ratio of area of square to area of circle is $\frac{\pi}{4}$.

Figure would look something like this



Let us generate n 2-dimensional random points with the constraint that these points lie within the square. Let x be the number of points among these n points, which lie in the circle as well. As we assume that the random numbers have equal probability of falling at every point in the square, thus we can estimate ratio of their areas as $\frac{x}{n}$. Therefore,

$$\frac{x}{n} = \frac{\pi}{4}$$

$$=> \pi = \frac{4x}{n}$$

. In this way, we can find the approximate value of π.

3.3 Monte Carlo Integration

Let us name the function we want to integrate as $f(x)$ and limits as a to b. Thus,

$$f(x) = e^{\frac{-x^2}{2}}$$

$$a = 0, b = 1$$

We generate n random numbers with a constraint that they must lie in the range $[a, b]$. We plug in each of the random numbers in the function $f(x)$ and get their average. We assume that the random number generated have equal probability of falling in all the points in the range. Thus multiplying them with $b - a$ will give us the expected value of the integral.

4 Implementational Flow

We have implemented the above explained 4 methods of parallel random number generation and tested them for the 2 applications of Monte Carlo simulation. Supposing that the total number of processes are P, we use $N = 10^5$ random numbers in our simulation. Then, each process generates $\frac{N}{P}$ numbers in parallel, using one of the algorithms. Now,

1. If we want to calculate value of π, then each process generates twice the number than mentioned above (one for each coordinate in 2D plane). They calculate how many of these points are inside circle and return that value to Master Process 1. Process 1 then calculates approximate value of π as described above.

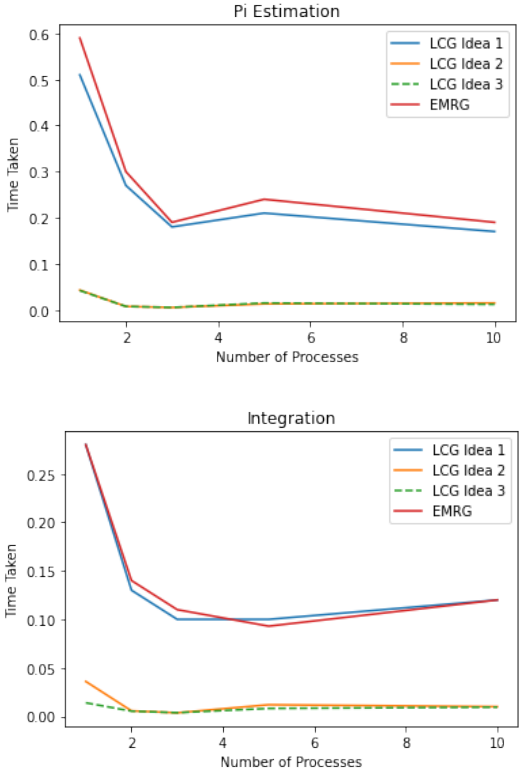
2. If we want to calculate value of the integral $\int_0^1 e^{-\frac{x^2}{2}} dx$, then each process gets the sum of values of $f(x)$ where $f(x) = e^{\frac{-x^2}{2}}$ by substituting x by the random numbers generated. They then return this value to Master Process 1. Process 1 then calculates final value using method explained above.

5 Analysis

Following table summarizes the results for all combinations of random number generators and monte carlo simulations for various degree of parallelism.

Notations used are :
For Random Number Generators
1 - Parallel LCG - Idea 1
2 - Parallel LCG - Idea 2
3 - Parallel LCG - Idea 3
4 - Ecuyer's Multiple Recursive Generator
For Monte Carlo Simulation
1 - Approximation of π - True value = 3.141592653
2 - Approximation of $\int_0^1 e^{-\frac{x^2}{2}} dx$ - True value = 0.8566068016
p is number of processes used

Random Number Generator	Monte Carlo Simulation	Approximate Value	Time Taken in seconds				
			p=1	p=2	p=3	p=5	p=10
1	1	3.13937	0.51	0.27	0.18	0.21	0.17
1	2	0.8558022	0.28	0.13	0.1	0.1	0.12
2	1	3.151706	0.043	0.0075	0.0053	0.013	0.015
2	2	0.8565508	0.036	0.0057	0.0037	0.012	0.01
3	1	3.14129	0.042	0.0075	0.0053	0.015	0.012
3	2	0.8553136	0.014	0.0054	0.0038	0.0082	0.0095
4	1	3.13721	0.59	0.3	0.19	0.24	0.19
4	2	0.855868	0.28	0.14	0.11	0.093	0.12



From the above graph we observe that there is a drastic dip in the time taken as we increase the number of processes, initially. However as the number of processes are further increased, the time taken increases a bit and remains somewhat constant. This may be attributed to the fact that we should also take into account, the heavy toll due to time taken by process creation and message passing. We observe that about 5 number of processes are ideally the best case for maximizing the benefit of parallelization.

6 Conclusion

This project brought to light the importance of joint parallelism where each process does their part and contributes together to solve a single problem. We understood in depth functioning of the Message Passing Interface and used it to our benefit in solving a single problem by putting in the joint efforts of multiple processes. Exploring the different techniques for Random Numbers we got to learn and devise our own algorithms to parallelize them. We also got to explore a range of stochastic probabilistic algorithms like Monte Carlo to solve a continuous range defined math problem. Finally we conclude by saying that it was an enriching experience to actually model a distributed system and put it to use for solving real life problems.