Parallel Monte Carlo Simulation obeying various Random number generation Techniques

Sayantan Jana, Prajwal Krishna Maitin, Samudraneel Dasgupta

Mentor : Additya Popli

1 Background

Monte Carlo (MC) methods are a subset of computational algorithms that use the process of repeated random sampling to make numerical estimations of unknown parameters. They allow for the modeling of complex situations where many random variables are involved, and assessing the impact of risk. Now how do they actually estimate ?

> "As the number of identically distributed, randomly generated variables increases, their sample mean (average) approaches their theoretical mean. "

This is significantly the basis for Monte Carlo simulations and allows us to build a stochastic model by the method of statistical trials. Based on the result of each trial, a mean result can actually be built, which though is actually the sample mean but as the theorem states approaches the theoretical mean if we run a large number of simulations. Now to achieve a better result, as the demand for simulation increases, parallelism actually comes to use which enables one to run as many simulations as one want on multiples cores in significantly less time than the sequential simulations, producing the same result without any chances in error in estimate obtained .

Random number generators 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. PRNGs are central in applications such as simulations (e.g. for the Monte Carlo method), electronic games (e.g. for procedural generation), and cryptography. Random number generators now come to great effect in developing Monte Carlo-method simulations, as debugging is facilitated by the ability to run the same sequence of random numbers again by starting from the same random seed. Hence anyone willing to produce any result through Monte Carlo methods take care of the random number generation as it requires the samples to be as random as possible and also following certain uniformity. A look can be taken into Chapter 1 : Uniform Random Number Generation of Handbook of Monte Carlo Methods for detailed techniques of generation. We have particularly sampled a few to use, obviously the ones that offer possibility of parallelism.

2 Project Goal, Applications

We aim to show that the area for any bounded convex shape can be estimated using Monte Carlo Simulation . We necessarily aim to explore shapes in only 2 dimensions. The only requirement is probably the availability of a test function for checking given any point

whether it lies inside the shape or not. Also if we do possess an analytical or mathematical way of figuring out the area, we can make comparision checks between the given areas, one estimated by Monte Carlo method, other given by the analytical formula .

Since the requirement of test function isn't well suited for all convex shapes, we focus our attention on 2 shapes particularly, first , a convex polygon and second, a regular ellipse.

Now apart from running Monte Carlo simulations, our goal is also to explore how do various random number generations behave for requirement of large length of sequences. Now as we learnt in above section, that there is a need to run a large no. of simulations, we essentially exploit parallelism through the use of MPI . A needful constraint of parallelism is that area on same no. of simulations obtained using single core vs multicore must be essentially same. Now how do we obtain parallelism in general is discussed in further sections .

3 Random number generation techniques

The techniques we explore mostly fall under 2 categories. First, **Linear congruential generator** (LCG), that yields a sequence of pseudo-randomized numbers calculated with a discontinuous piecewise linear equation. Second, **Multiple-Recursive Generators** (MCG), an alternative to explore generalizations of LCGs that had high periods but could still be implemented with single-precision floating-point arithmetic on 32-bit computers basically aiming to make a LCG have any order of magnitude . Though few of them still need 64 bit computers.

- **Lehmer random number generator :** A type of linear congruential generator (LCG) that operates in multiplicative group of integers modulo n. The general formula is :

  $$X_{k+1} = a \cdot X_k \mod m$$

  where the modulus m is a prime number or a power of a prime number, the multiplier a is an element of high multiplicative order modulo m (e.g., a primitive root modulo n), and the seed X_0 is coprime to m.

- **BSD libc LCG :** BSD libc is the core library for the C language with the BSD operating systems. The task is to replicate historic rand() function from BSD libc. A general Linear Congruential generator is defined by recurrence relation:

  $$X_{n+1} = (aX_n + c) \bmod m$$

  where X is the sequence of pseudorandom values, and
  $m, 0 < m-$ the "modulus"
  $a, 0 < a < m-$ the "multiplier"
  $c, 0 \leq c < m-$ the "increment" $X_0, 0 \leq X_0 < m-$ the "seed" or "start value"
  BSD formula :

  $$X_{n+1} = 1103515245 \times X_n + 12345 (\mathrm{mod} 2^{31})$$

  $$X_n \in [0, 2147483647].$$

- **MRG given by L'Ecuyer, Blouin and Couture (1993) :** A general Multiple Recursive Generator is defined by recurrence relation :

  $$z^{[k]} = a_1 z^{[k-1]} + a_2 z^{[k-2]} + \ldots + a_m z^{[k-m]} (\mathrm{mod} \eta)$$

  $$u^{[k]} = z^{[k]/\eta}$$

for some positive integer k, positive integers $a_1, a_2, \ldots, a_m$, and seed values $z^{[0]}, z^{[1]}, \ldots, z^{[k-1]}$. Through judicious selection of parameters, an MRG can have a period as high as $\eta^k - 1$. As a generalization of LCGs, MRGs share many of the same properties. They have similar lattice structures, and certain MRGs exhibit strong correlations between pseudorandom numbers separated by large lags.
L'Ecuyer, Blouin and Couture (1993) performed an extensive search for good MRGs. One that they found is :

$$z^{[k]} = 1,071,064 \times z^{[k-1]} + 2,113,664 \times z^{[k-7]} \left(\mod \left(2^{31} - 19\right)\right)$$

This has period $\eta^7 - 1 \approx 2^{217}$ and exhibits generally good lattice properties for dimensions up to 20. It can be efficiently implemented on a 32 -bit computer, although this is less of a concern now that 64 -bit computers are common.

- **L'Ecuyer's Multiple Recursive Generator MRG32k3a :** L'Ecuyer studied Combined Multiple Recursive Generators (CMRG) in order to produce a generator that had good randomness properties with a long period, while at the same time being fairly straightforward to implement. The best known CMRG is MRG32k3a which is defined by the set of equations :

$$y_{1,n} = (a_{12} y_{1,n-2} + a_{13} y_{1,n-3}) \bmod m_1$$
$$y_{2,n} = (a_{21} y_{2,n-1} + a_{23} y_{2,n-3}) \bmod m_2$$
$$x_n = (y_{1,n} + y_{2,n}) \bmod m_1$$

for all $n \geq 3$ where

$$\begin{array}{lll} a_{12} = 1403580 & a_{13} = -810728 & m_1 = 2^{32} - 209 \\ a_{21} = 527612 & a_{23} = -1370589 & m_2 = 2^{32} - 22853 \end{array}$$

- **Chiang's Multiple Recursive Generator :** Multiple Recursive Generator (MRG) has been considered by many scholars as a very good Random Number generator. Sequential search can be applied to identify the MRGs of orders one, two, and three which are able to produce RNs with good lattice structure in terms of the spectral value and Beyer quotient. In approximately 19.3 billion candidates, Chaing et al found that only four MRGs, namely, (1280550, -45991), (0,45991,1758790), (885300443,0,1552858447), and (885300443, 1546795921,598295599), have passed all the theoretical and empirical tests. We will use the last one, defined by the set of equations :

$$z^{[k]} = 885300443 \times z^{[k-1]} + 1546795921 \times z^{[k-2]} + 598295599 \times z^{[k-3]} \left(\mod \left(2^{31} - 1\right)\right)$$

This MRG has a period of about 2^{93} which certainly meets the minimal requirement stated by L'Ecuyer, and also possesses excellent lattice structure.

Note for our purpose we need uniform random numbers . The sequence of integers $X_0, X_1, X_2, X_3, ...$ are the output of the generator. When divided by the corresponding modulo, they give pseudo-random outputs with a uniform distribution on the unit interval [0, 1) . These may then be transformed into various other distributions.

4 Parallelising random number generation

As we have picked almost all the algorithms, that follow certain sort of recursive generation , we can actually reach any particular element of sequence without needing to find all the elements in between, but just estimating a few. The simple skip ahead strategy whereby each process has its own copy of state and produces a contiguous segment of the sequence independently of all other process works well, provided we have a very efficient way to skip ahead an arbitrary number of points. Now let's discussion how do we really parallelise.

At any point in the sequence the state can be represented by a vector or a pair of vectors (as needed in MRG32k3a). Let's focus on how we parallelise MRG32k3a at the moment, the rest follow a same way.

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

for $i = 1, 2$. It follows that the two recurrences in above can be represented as

$$Y_{i,n+1} = A_i Y_{i,n} \mod m_i$$

for $i = 1, 2$ where each A_i is a 3×3 matrix, and therefore

$$Y_{i,n+p} = A_i^p Y_{i,n} \mod m_i, p \geq 0$$

To efficiently compute A_i^p for large values of p , one can use the classic "divide and conquer" strategy of iteratively squaring the matrix A_i. A well known technique known as Matrix Exponentiation. Let's say,

$$p = \sum_{j=0}^{k} g_j 2^j$$

where $g_j \in \{0, 1\}$ and just look at these elements of the sequence

$$A_i, A_i^2, A_i^4, A_i^8, A_i^{16}, \ldots, A_i^{2^k}, \quad \mod m_i$$

These can be obtained by successively squaring the previous term found. It is then a simple matter to compute

$$A_i^p Y_i = \prod_{j=0}^{k} A^{g_j 2^j} Y_i \mod m_i$$

for $i = 1, 2$ and the entire process can be completed in approximately $O(\log_2 p)$ steps.

A further improvement :
We can improve the speed of this procedure at the cost of more memory by expanding the exponent p in a base higher than 2 so that

$$p = \sum_{j=0}^{k} g_j b^j$$

for some $b > 2$ and $g_j \in \{0, 1, \ldots, b-1\}$. This improves the "granularity" of the expansion. To illustrate, suppose $b = 10$ and $p = 7$. Then computing $A^p Y$ requires only a single lookup from memory, namely the pre-computed value A^7, and a single matrix-vector product.

However if $b = 2$ then $A^7Y = A^4\left(A^2(AY)\right)$ requiring three lookups and three matrixvector products.

Now following the above method, using a base seed we pass it to the 1st process and for all other processes we estimate the $N - th, 2N - th, 3N - th..$ elements of the sequence in each process and use it as a seed for generation of a continuous N block of random numbers, hence having generated $N * numprocs$ random numbers parallely quite faster than having generated these random numbers in sequential manner.

5 Implementation, Test Generation and Correctness measure

We have kept all our experiments to 2D for this project, so we first declare a structure for 2D points and then we have developed a class for lines and another for polygons. What we wanted to showcase was that our method can estimate area of any convex polygons up to a reasonable accuracy.

For generating random convex polygons, we first randomly sample N points then we find the convex hull of these N points. This way we get a convex polygon randomly. While sampling points we noted that if we were deriving x and y coordinates from same instant of generators, it was noted that the accuracy of estimated area was poor, hence we shifted to sample each dimension from an independent distribution. We can explain this as in the prior case, there was a correlation between x and y coordinates, hence it was not an iid, which is assumed by the Monte Carlo Method.

We further calculated the areas of polygons analytically as we wanted to compare the answer produced by Monte Carlo method. For this we leveraged the fact that the centroid of a convex polygon lies inside the polygon, hence we divide the polygon of n sides into n triangles each with a side as base and centroid as the third vertices, then we used area of triangle formula to calculate area of each of these triangles and their sum as area of the polygon. Given the coordinates of the three vertices of any triangle, the area of the triangle is given by:

$$\left|\frac{A_x(By - Cy) + Bx(Cy - Ay) + Cx(Ay - By)}{2}\right|$$

where A_x and A_y are the x and y coordinates of the point A etc.

Further, we need to make a utility function, which can cheaply tell if a given sample point lies within the polygon or not. For this we again harnessed that centroid of a convex polygon lies inside a polygon, this way we found equation of each of the side of polygon, in such as way that for a point to lie inside the polygon it should lie on the negative side to all lines. This gives us quick test for if points lie inside the polygon or not. This test is required to be fast enough as it will be called in each iteration of the montecarlo sampling, and typically we do 10,000,000 runs. We could have optimized this step from O(n) to O(log n) where n is number of sides in polygon, however, we found that for small n in our case less than 20 it did not matter that much. All vertices of our polygon is within [0,1].

Now for the actual part, we calculate the area of the polygon as follows - 1. For each sample we generate a random points using various methods of random number generator and making sure that x and y have different seeds so that they are not correlated. The x and y both lie in range [0,1]. 2. Next we test if this points lie inside the polygon or not, if it does we increase the hit number by 1. 3. We can perform each of these samples independently, and we divide run 4 samples in parallel. 4. At the end we get M number of samples per process, then we combine the results of each of these processes getting 4M

samples, and calculate the area. 5. Since our samples are in range [0,1], they can cover an area of 1 unit square, now number of points which lied within polygon by total number of samples * (1 unit square) gives us the area.

Area of Polygon = $\frac{S}{N}$ Unit squares
where S is the no. of samples found inside polygon and N is the total no. of samples

We could have done similar work for any other arbitrary shape, if we get 2 functions for the shape, a quick test to find if a point P lies within the shape and second a method for analytically calculating area of the polygon for comparison of the results. For getting estimation of the area, just first function is sufficient.

As we have done for a polygon we do the same for an ellipse as well, we compare the area computed by Monte Carlo Simulation with the area obtained by area of ellipse obtained by Formula :
Area of the Ellipse = $\pi r_1 r_2$
where r_1 and r_2 are length of semi major axis and semi minor axis of the ellipse.

6 Workflow : Algorithm

- Process 0 reads the input and broadcasts the entire data to each of other processes.
- Process 0 also broadcasts 2 seeds S_x and S_y for coordinate x and coordinate y.
- Each process knows the number of simulations it has to run, same for all the processes, let's say N, so each process on receiving the seeds, generates A_{rank*N}-th element of sequence using the Matrix Exponentiation strategy and uses that as their individual seeds. Get the notion that A_0-th element is basically the initial two seeds S_x and S_y.
- Now each of the process run their set of simulations, generating 2 random numbers coordinate x and coordinate y in each iteration and uses them as the sample for checking if it is inside the shape, either the convex polygon or the regular ellipse and finally finds the total number of hits after going through their assigned no. of samples. Note that we generate random numbers on the fly, i.e., we do not store them anywhere else it would again lead to surge in memory complexity.
- Using MPI Allreduce function to sum up the hits , the process 0 gets to know the total no. of hits out of the total no. of trials across the processes, hence process 0 finally computes the area as hit ratio.

7 Complexity

Let's say we run a total of M simulations across all processes, the number of processes being P on a polygon having N sides then,

For 1 Core (1 process) :

Time Complexity : $O(N * M)$,
with the check if a point lies inside a polygon or not distributed taking $O(N)$ time .

Memory Complexity : $O(N)$,
random number generation can be assumed to have used a constant memory space.

For multicore (say P processes) :

Time Complexity : $O(\frac{M*N}{P})$,
with the check if a point lies inside a polygon or not distributed taking $O(N)$ time .

Memory Complexity : $O(N * P)$,
random number generation can be assumed to have used a constant memory space.

8 Results

8.1 Area of convex polygon :

Here is the result of various random generation ways for 4 sampled input files out of the testset we created (can be found in the github repository) for the application pf finding area of polygon for various no. of samples, i.e, 10^5, 10^6, 10^7, for 1 core vs 4 cores. Performance is seen in terms of the error rate (obatined as compared to actual area obtained by analytical method) and the time taken by the code in each case. Error is stated in % and time in seconds.

1 core vs 4 core performance of Monte Carlo simulation using Lehmar's LCG :

	error[10^5]	time[10^5]	error[10^6]	time[10^6]	err[10^7]	time[10^7]
1.in	0.0876	0.01 — 0.004	0.0061	0.09 — 0.04	0.0076	0.87 — 0.24
2.in	0.3870	0.01 — 0.003	0.0687	0.09 — 0.04	0.0171	0.86 — 0.23
3.in	0.1396	0.01 — 0.005	0.0245	0.10 — 0.03	0.0036	0.95 — 0.25
4.in	0.0639	0.01 — 0.004	0.0460	0.08 — 0.02	0.0221	0.82 — 0.20

1 core vs 4 core performance of Monte Carlo simulation using BSD libc LCG :

	error[10^5]	time[10^5]	error[10^6]	time[10^6]	err[10^7]	time[10^7]
1.in	0.4731	0.02 — 0.004	0.0502	0.09 — 0.04	0.0007	0.86 — 0.25
2.in	0.0546	0.01 — 0.003	0.0874	0.09 — 0.04	0.0075	0.85 — 0.21
3.in	0.2768	0.01 — 0.005	0.1132	0.10 — 0.03	0.0066	0.94 — 0.24
4.in	0.3376	0.01 — 0.004	0.0549	0.08 — 0.03	0.0072	0.82 — 0.20

1 core vs 4 core performance of Monte Carlo simulation using MRG given by L'Ecuyer, Blouin and Couture :

	error[10^5]	time[10^5]	error[10^6]	time[10^6]	err[10^7]	time[10^7]
1.in	0.29	0.027 — 0.007	0.009	0.16 — 0.09	0.04	1.62 — 0.45
2.in	0.24	0.016 — 0.007	0.013	0.15 — 0.09	0.05	1.58 — 0.43
3.in	0.07	0.025 — 0.008	0.055	0.17 — 0.10	0.06	1.67 — 0.47
4.in	0.13	0.01 — 0.006	0.0549	0.18 — 0.08	0.0095	1.66 — 0.44

1 core vs 4 core performance of Monte Carlo simulation using L'Ecuyers MRG32k3a :

	error[10^5]	time[10^5]	error[10^6]	time[10^6]	err[10^7]	time[10^7]
1.in	0.075	0.026 — 0.008	0.095	0.2 — 0.05	0.028	1.9 — 0.53
2.in	0.181	0.019 — 0.005	0.123	0.19 — 0.06	0.024	1.88 — 0.51
3.in	0.213	0.027 — 0.008	0.114	0.20 — 0.057	0.024	1.97 — 0.53
4.in	0.169	0.022 — 0.006	0.136	0.18 — 0.07	0.021	2.06 — 0.53

1 core vs 4 core performance of Monte Carlo simulation using Chiang's MRG :

	error[10^5]	time[10^5]	error[10^6]	time[10^6]	err[10^7]	time[10^7]
1.in	0.18	0.036 — 0.009	0.022	0.34 — 0.09	0.008	3.4 — 0.93
2.in	0.051	0.045 — 0.011	0.16	0.33 — 0.10	0.028	3.38 — 0.92
3.in	0.052	0.039 — 0.014	0.14	0.36 — 0.09	0.021	1.52 — 0.94
4.in	0.078	0.044 — 0.013	0.07	0.39 — 0.11	0.07	3.24 — 0.81

8.2 Area of a regular ellipse :

Here is the result of various random generation ways for a particular sampled input file out of the testset we created (can be found in the github repository) for the application pf finding area of polygon for various no. of samples, i.e, 10^5, 10^6, 10^7, for 1 core vs 4 cores. Performance is seen in terms of the error rate (obatined as compared to actual area obtained by analytical method) and the time taken by the code in each case. Error is stated in % and time in seconds.

1 core vs 4 core performance for each of the Monte Carlo simulation using different random generation techniques :

	error[10^5]	time[10^5]	error[10^6]	time[10^6]	err[10^7]	time[10^7]
Lehmer RNG	0.4731	0.02 — 0.004	0.2815	0.03 — 0.01	0.0142	0.32 — 0.10
BSD libc LCG	0.1213	0.003 — 0.001	0.4497	0.03 — 0.009	0.2313	0.277 — 0.07
Couture MRG	1.6493	0.01 — 0.004	0.0690	0.11 — 0.03	0.1107	1.04 — 0.27
L'Ecuyer's MRG	0.1065	0.02 — 0.007	0.6681	0.15 — 0.05	0.0190	1.46 — 0.38
Chiang MRG	1.1033	0.03 — 0.01	0.1575	0.30 — 0.09	0.0795	2.90 — 0.74

9 Conclusion

Hence, we found it to be a very powerful tool, specially in case of calculating area(and subsequently integral) of functions for which we cannot calculate analytically. Also we can harness it to calculate area of functions for which we know analytical methods, but is not computationally cheap. This method provides us flexibility of increasing accuracy and precision as per our requirement, if we require area with lesser accuracy we can get it more cheaply.

For code refer to Github :
`https://bit.ly/2KtlfYF`

References

[1] Wikipedia : Lehmer random number generator :
`https://bit.ly/3eOPyHt`

[2] Rosettacode : Linear Congruential Generator :
`https://bit.ly/3cxbm8q`

[3] Multiple Recursive Generators :
`https://bit.ly/2W43LaX`

[4] Parallelisation Techniques for Random Number Generators :
`https://bit.ly/3cJBKf9`