Patrick Byrne
CS475 Spring 2020
byrnepat@oregonstate.edu

Project 1
OpenMP: Monte Carlo Simulation

The Monte Carlo simulation is a type of computational algorithm that is used to model the probability of various outcomes that are not easily predictable due to randomness. Given a set of parameters, a Monte Carlo simulation can determine the range of possible outcomes and create a probability distribution showing the likelihood of each outcome. These simulations involve the use or random numbers to solve problems that might be deterministic in nature, relying on repeated random sampling to produce results and derive inference.

The Experiment

Our project involves a Monte Carlo simulation where a laser beam, aimed at a 30 degree angle, is shot at a circle. If the beam hits the circle, it bounces off. Underneath the circle is an infinite plate. The simulation attempts to determine the probability that the beam will bounce in a downward angle and hit the plate. The circle randomly changes both its location (from the origin) and its size.

The goal of the simulation is to calculate the probability that the circle is positioned correctly and the beam hits the plate. The probability is calculated by dividing number of hits per run by the number of trials per run. Hitting the plate is dependent on the coordinates of the center of the circle and the circle's radius. Since these three values are randomly determined each time we run the program, the probability for a given set of xc, yc, and r values needs to be calculated many times to approach the actual probability. The more times you run the simulation, the more likely you are to get the actual probability. This is where parallelization comes into play. By using openMP and running parallel threads, we can do more calculations and arrive at a reasonable candidate for actual probability much faster than if we were running a single thread.

Another very important aspect of this project is to calculate the maximum performance of the machine we are using to determine the probability. The performance is a measure of how many MegaTrials (millions of trials) the machine can perform each second (MegaTrials/sec). The maximum performance is calculated by running a nested for loop. The outside for loop determines how many times the program runs for each set of trials. The inside for loop runs the individual trials to determine both probability and maximum performance. I used 1000 runs for every set of trials (NUMTRIES = 1000).

I ran my simulations on an AMD Ryzen 7 3800x 8 core processor. Because it had 8 cores, I used 8 for the number of threads in my program. My script runs the program and gathers data for each possible number of threads from 1 to 8. We would expect that, as the number of cores used increases, the better the maximum performance. I kept my threads number within the maximum number of cores of my machine in the hopes that it would more accurately reflect this expectation.

The number of trials used to calculate the probability values ranges from 1 to 1,000,000 (increasing by powers of 10). Each set of trials is repeated with 1000 runs (NUMTRIES = 1000). This value was kept constant throughout the entire experiment. Both the probability and the performance were calculated for each trial. The performance is a measure of how many trials (or millions of trials (megatrials) in this case) we can computer per second. We should expect to see that using parallel threads with openMP will increase performance.

Estimating the Actual Probability

My data shows the actual probability to be 0.13 (to two decimal places).

The following set of tables show the trials, probabilities, and maximum performances (in MegaTrials/sec) for each of the 8 thread runs. Unlike in the performance graphs below, the probability for each run can clearly be seen here. For every thread, the probability is 0.13 at the maximum number of trials. This demonstrates the need for many trials to calculate an accurate probability. It also shows that calculating an accurate probability is completely independent of the number of threads. The threads allow you to calculate it faster, but do not improve accuracy. NUMTRIES = 1000 for each set of threads/trials.

Threads = 1

Trials	Probab	MaxPerf
1	0	1.67
10	0	12.5
100	0.14	37.04
1000	0.14	45.25
10000	0.13	39.17
100000	0.13	38.27
1000000	0.13	37.73

Threads = 2

Trials	Probab	MaxPerf
1	0	0.91
10	0.1	9.09
100	0.12	35.71
1000	0.13	87.72
10000	0.14	78.31
100000	0.13	76.64
1000000	0.13	74.35

Threads = 3

Trials	Probab	MaxPerf
1	0	0.56
10	0.1	5.26
100	0.09	66.67
1000	0.12	116.28
10000	0.13	125
100000	0.13	113.11
1000000	0.13	108.54

Threads = 4

Trials	Probab	MaxPerf
1	0	0.59
10	0.1	5.56
100	0.18	40
1000	0.14	142.86
10000	0.13	167.5
100000	0.13	148.77
1000000	0.13	144.65

Threads = 5

Trials	Probab	MaxPerf
1	0	0.56
10	0	4.76
100	0.08	41.67
1000	0.12	169.49
10000	0.13	207.9
100000	0.13	186.43
1000000	0.13	180.63

Threads = 6

Trials	Probab	MaxPerf
1	0	0.5
10	0.1	4.76
100	0.07	43.48
1000	0.1	188.68
10000	0.13	251.89
100000	0.13	222.92
1000000	0.13	216.51

Threads = 7

Trials	Probab	MaxPerf
1	0	0.5
10	0	4.55
100	0.15	41.67
1000	0.14	200
10000	0.13	287.36
100000	0.13	261.37
1000000	0.13	251.04

Threads = 8

Trials	Probab	MaxPerf
1	0	0.48
10	0.3	4.17
100	0.23	40
1000	0.14	196.08
10000	0.13	326.8
100000	0.13	298.86
1000000	0.13	281.94

Performance Graphs

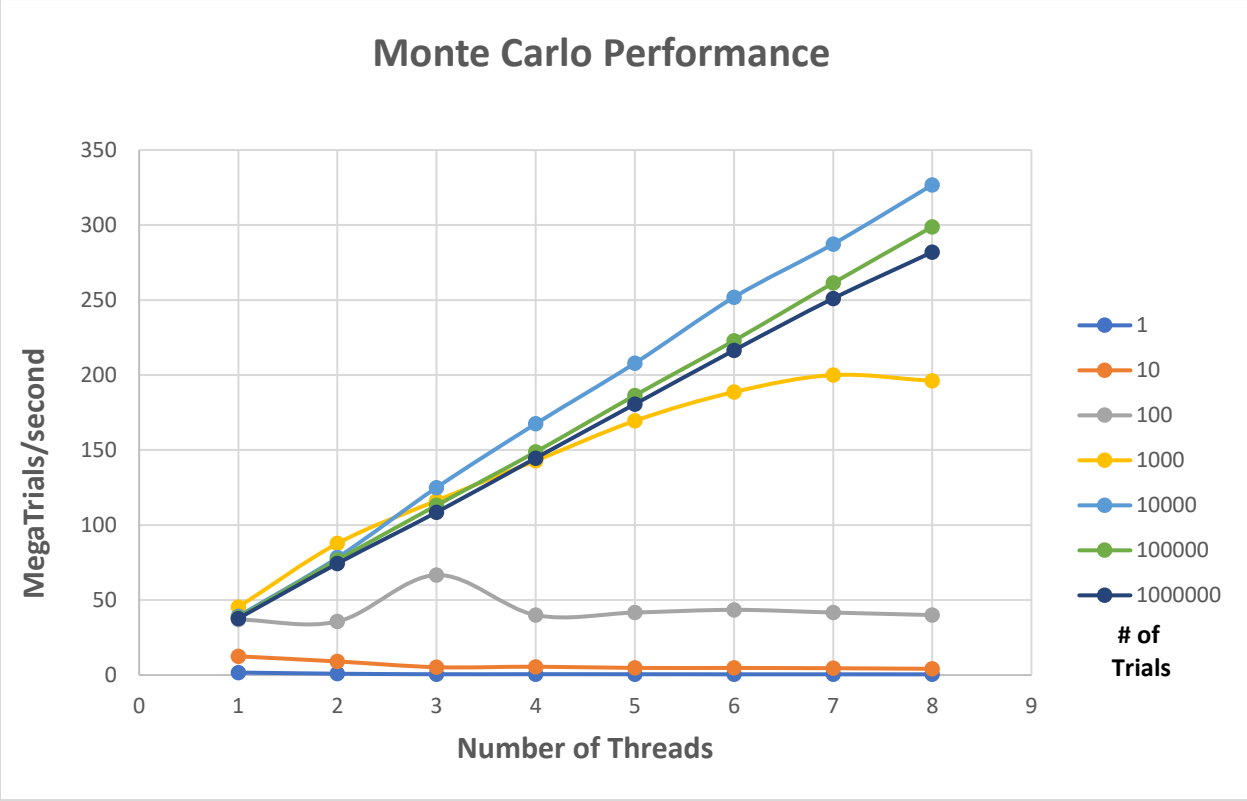
This is the table I used for all graphs. The top row represents the number of trials used (from 50 to 1000000). The 1st column represents the number of threads used (1,2,4,8,12, or 16). The maximum performance was tested by number of trials and by number of threads.
Reading the table by column shows the maximum performance with number of threads the independent variable and the number of trials kept constant.
Reading the table by row shows the max performance with number of trials the independent variable and the number of threads kept constant.
Each set of trials was run 1000 times (NUMTRIES = 1000).

Number of Trials

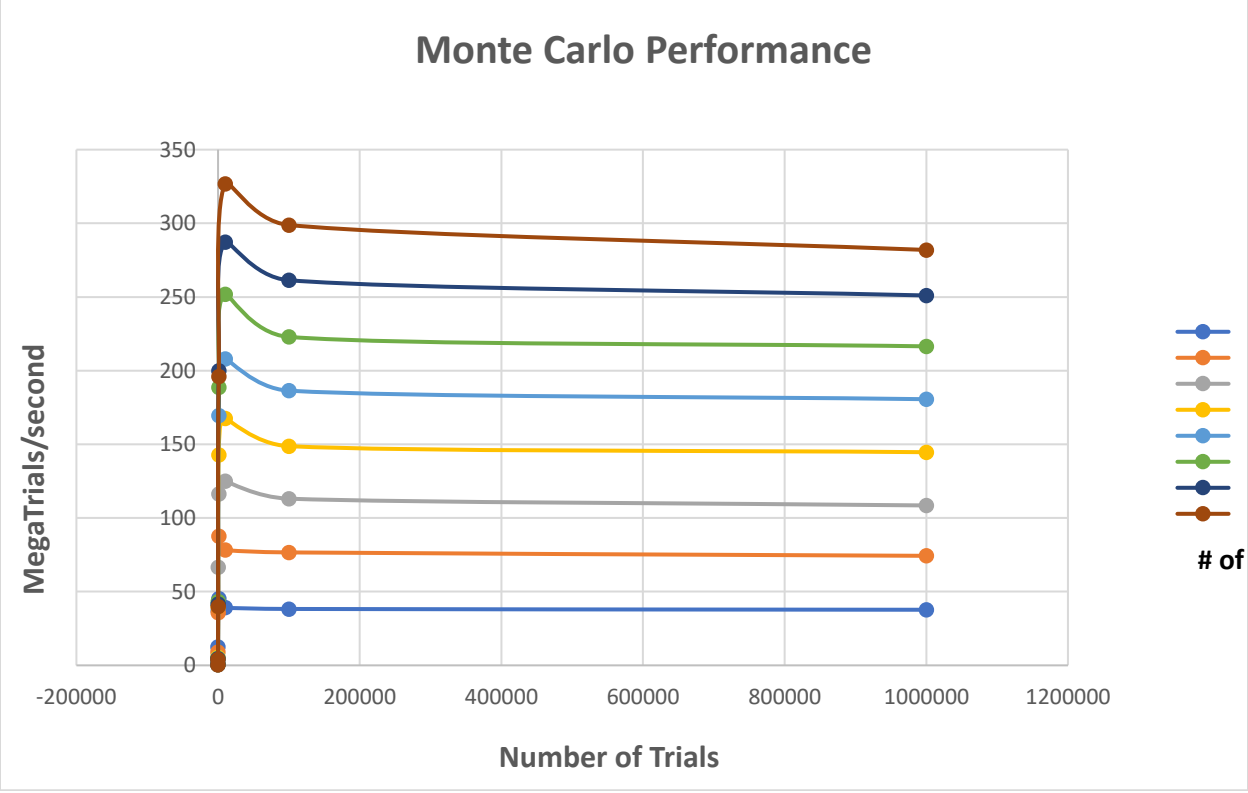
Number of Threads	1	10	100	1000	10000	100000	1000000
1	1.67	12.5	37.04	45.25	39.17	38.27	37.73
2	0.91	9.09	35.71	87.72	78.31	76.64	74.35
3	0.56	5.26	66.67	116.28	125	113.11	108.54
4	0.59	5.56	40	142.86	167.5	148.77	144.65
5	0.56	4.76	41.67	169.49	207.9	186.43	180.63
6	0.5	4.76	43.48	188.68	251.89	222.92	216.51
7	0.5	4.55	41.67	200	287.36	261.37	251.04
8	0.48	4.17	40	196.08	326.8	298.86	281.94

To represent both ways of looking at the data, three graphs were used. I graphed performance vs the number of threads, performance vs the number of trials, and performance vs the number of trials using a log base 10 scale for the x-axis (number of trials). I used the log base 10 scale to give an even spacing between each separate number of trials. As can be seen in the first performance vs trials graph, the smaller number of trials values are compressed and difficult to analyze.

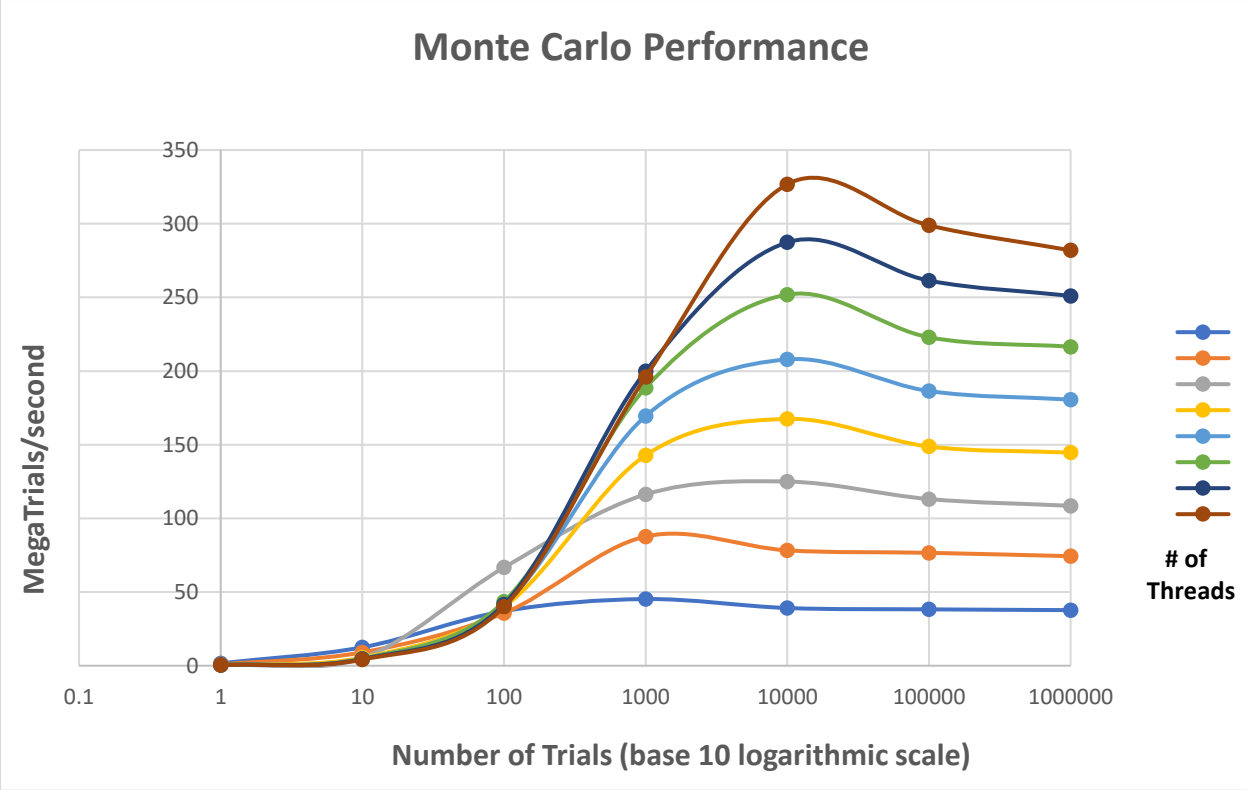
Performance vs Number of Threads



Performance vs Number of Trials



Performance vs Number of Trials (on a log base 10 scale)



From the tables and graphs above, we can clearly see that performance increases as the number of threads increases. This relationship looks linear for most of the thread runs but there appears to have been some external interference on the data when using 100 or 1000 threads. Perhaps some outside process had a detrimental influence on the program during those two runs.

Performance also increases with number of trials. However, the relationship is not linear. At a certain number of trials, the performance peaks for a given thread and then slightly declines as the number of trials increases after this point. For our experiment the maximum performance occurs at around 10,000 trials for 3-8 threads. It seems closer to 1000 trials for 1 or 2 threads. The earlier performance peak for 1 or 2 threads could be due to the lack of overhead needed for more threads. As the number of threads increases, the overhead required to set them up also increases. This could cause a delay in ramping up performance which may explain why the peak performance shows up later with more threads. Using the log base 10 scale for the number of trials made it much easier to see the performance peak since the smaller trial values spread apart and not compressed towards the beginning of the graph as the were in the standard performance vs number of trials graph.

Compute F_p, the Parallel Fraction

I calculated the Parallel Fraction F_p by comparing the 1-thread runs to the 8-thread run since 8 was the maximum number of threads. I also decided to do three different parallel fraction calculations, one for each of the three highest number of trials (10,000 trials, 100,000 trials, and 1,000,000 trials). I chose to do this because the maximum performance actually occurs at 10,000 trials instead of at the maximum number of trials. Performance slightly decreases from 10,000 to 1,000,000 trials. Meanwhile, the performance of 1 thread over the same number of trials had very little variation in it.

I thought it would be interesting to see the different parallel fractions among the three.

Table of Maximum Performance for three highest num of trials

Number of trials	Max Perf for 1 thread (in MegaTrials/sec)	Max Perf for 8 threads (in MegaTrials/sec)
10,000 trials	39.17	326.80
100,000 trials	38.27	298.86
1,000,000 trials	37.73	281.94

In order to calculate the parallel fractions, I needed to calculate the speedups using the following formula:
S = (Performance 8 Threads/Performance 1 Thread)

To calculate the parallel fraction, I used the following equation:
$F_p = (n/(n-1)) * (1-(1/S))$ where S is the speedup calculated earlier.

For 10,000 trials:

S = 326.80/39.17
S = 8.34

$F_p = (8/(8-1)) * (1-(1/8.34))$
$F_p = 8/7 * (1 – 0.120)$
$F_p = 8/7 * 0.880$
$F_p = 1.006$

For 100,000 trials:

S = 298.86/38.27
S = 7.81

$F_p = (8/(8-1)) * (1-(1/7.81))$
$F_p = 8/7 * (1 – 0.128)$
$F_p = 8/7 * 0.872$
$F_p = 0.997$

For 1,000,000 trials:

S = 281.94/37.73
S = 7.47

$F_p = (8/(8-1)) * (1-(1/7.47))$
$F_p = 8/7 * (1 – 0.134)$
$F_p = 8/7 * 0.866$
$F_p = 0.990$

The parallel fraction for 10,000 trials is greater than 1. This could be due to a process running in the background that would make the 1-thread run appear slower than it really was. This is also evident by the speedup which is 8.34. The speedup should not be larger than 8 which is the number of threads we used. However, if the 1-thread appears to run slower, than the ratio of performances could result in this type of erroneous value. The fraction was close to 1 however (1.006).

100,000 trials resulted in a parallel fraction of 0.997 and 1,000,000 trials resulted in a parallel fraction of 0.990 both of which are very close to 1.

A parallel fraction value close to 1 means that we are having very little in the way of diminishing returns. The program has a high level of efficiency. We are getting almost perfect parallelization from this experiment. The Monte Carlo program seems to be a simulation that is highly parallelizable.

Number of Threads	Number of Trials									
	50	100	500	1000	5000	10000	50000	100000	500000	1000000
1	20	37.04	46.73	46.3	40.39	39.32	38	38.14	37.56	37.6
2	33.33	50	83.33	78.74	59.03	80.39	74.9	74.72	74.22	72.98
4	38.46	58.82	106.38	114.94	119.9	168.92	151.1	147.71	145.82	144.04
8	19.23	35.71	116.28	161.29	224.22	268.82	311.53	301.11	291.46	284.13
12	16.67	26.32	108.7	178.57	277.78	327.87	402.58	392	391.73	377.19
16	13.89	27.78	102.04	188.68	362.32	421.94	503.02	521.1	511.4	503.47

Number of Trials as independent variable:
The data shows that for a give number of trials, the maximum

oes anyone have a resourcec on why lower number of trials resulted in poorer performance with increased cores
3 replies

Sherlock Jones[TA CS162] 18 hours ago
overhead on starting up the threads

fastkevin 18 hours ago
ah true

fastkevin 18 hours ago
thanks