Lachlan Sinclair

sinclala@oregonstate.edu

4/15/2020

CS 475 Spring

Project #1: Monte Carlo Simulation

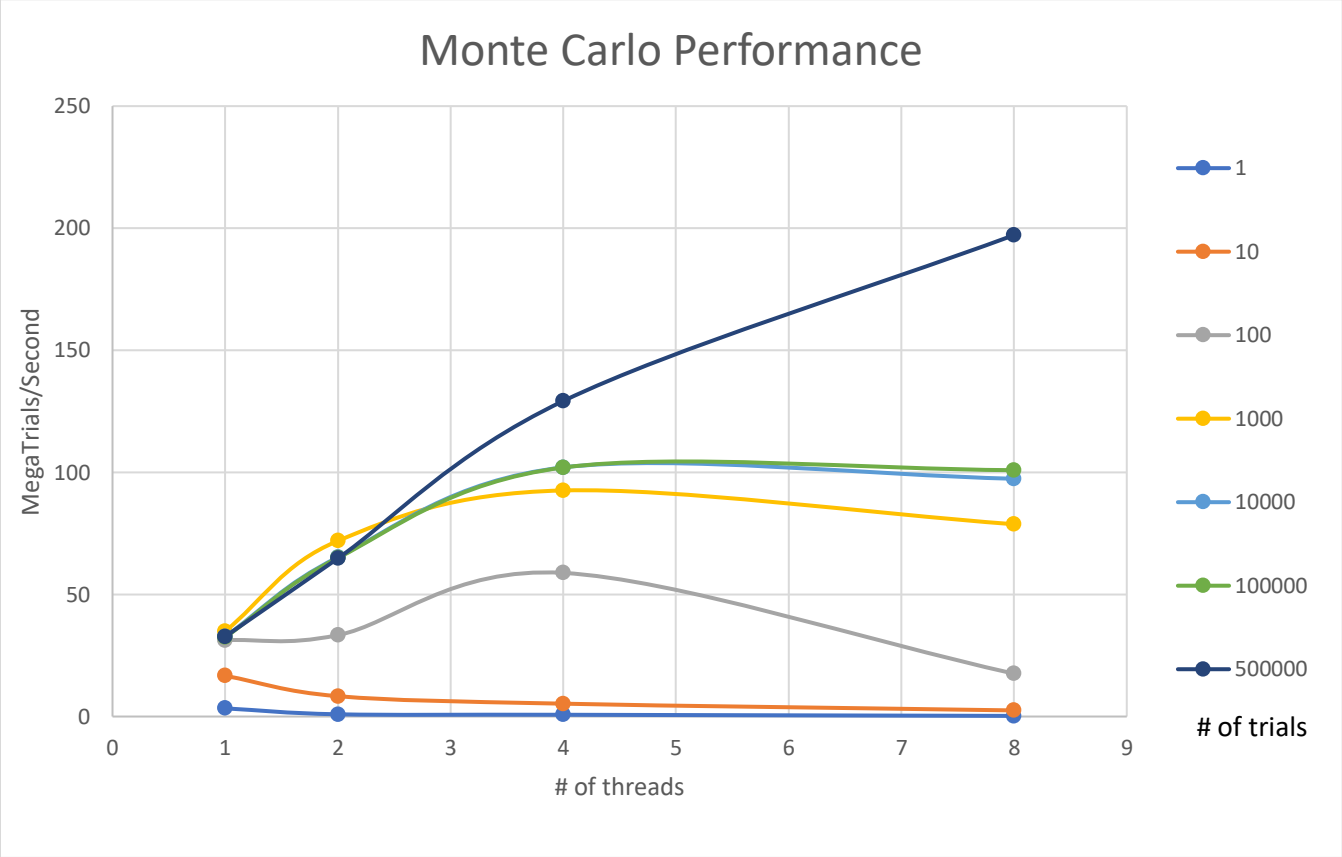
Results:

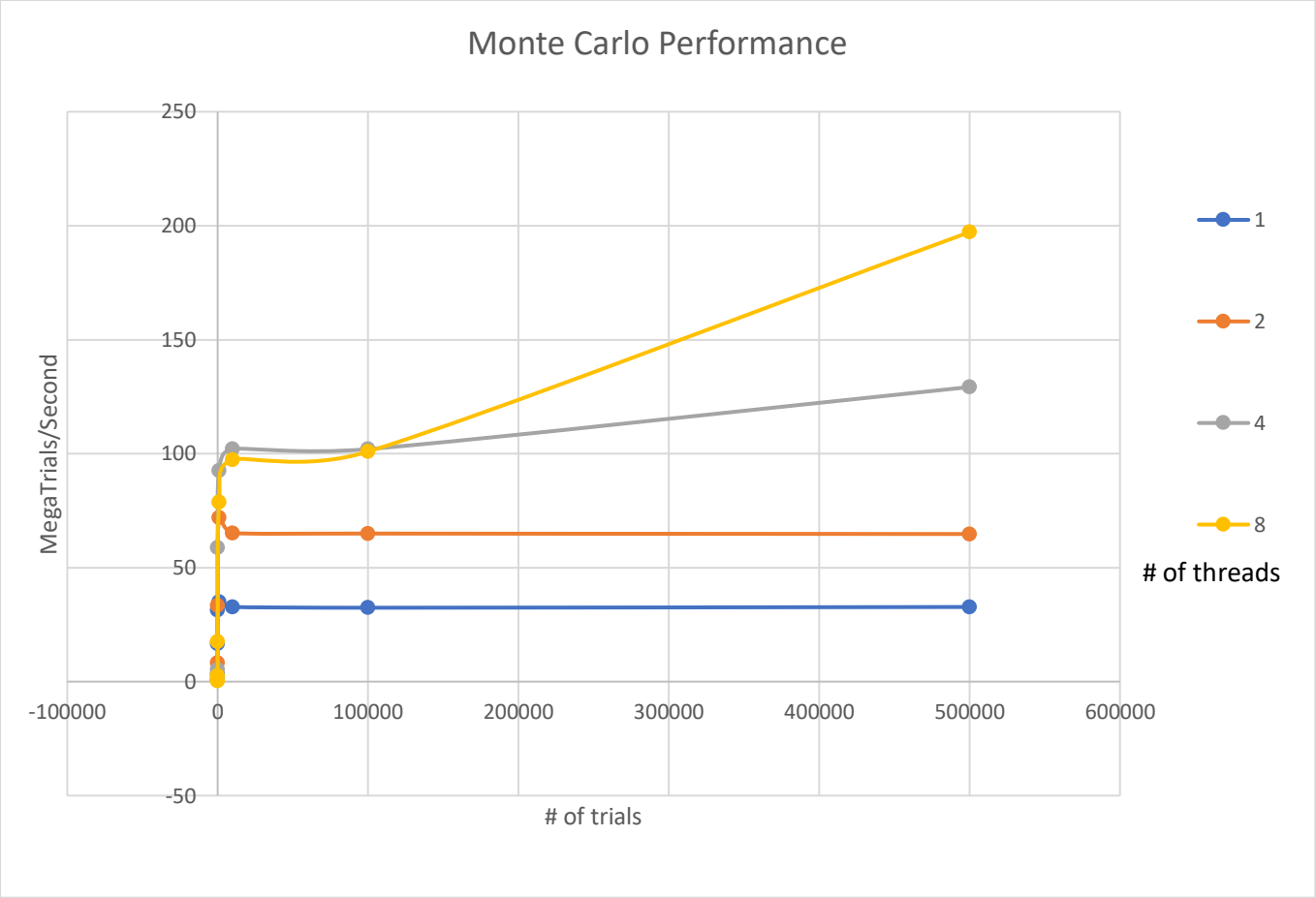
This is the excel formatted output obtained from the result.txt file that is generated by the python script project1script.py.

# of threads	# of trials	Probability	MegaTrials/Second
1	1	0	3.333308
1	10	0.3	16.66654
1	100	0.21	31.249762
1	1000	0.135	34.965054
1	10000	0.1311	32.7654
1	100000	0.12993	32.428577
1	500000	0.130628	32.750164
2	1	0	0.909084
2	10	0.1	8.33327
2	100	0.09	33.334373
2	1000	0.126	71.942497
2	10000	0.1315	65.231567
2	100000	0.12937	64.960373
2	500000	0.130628	64.769356
4	1	0	0.71428
4	10	0.1	5.263118
4	100	0.09	58.823082
4	1000	0.126	92.592888
4	10000	0.1315	102.040764
4	100000	0.12937	101.988777
4	500000	0.130628	129.232361
8	1	0	0.256408
8	10	0.1	2.499981
8	100	0.15	17.543726
8	1000	0.127	78.74028
8	10000	0.1321	97.371017
8	100000	0.13049	100.897995
8	500000	0.131368	197.184204

Below is the table used to generate the graphs:

	1	10	100	1000	10000	100000	500000
1	3.333308	16.66654	31.24976	34.96505	32.7654	32.42858	32.75016
2	0.909084	8.33327	33.33437	71.9425	65.23157	64.96037	64.76936
4	0.71428	5.263118	58.82308	92.59289	102.0408	101.9888	129.2324
8	0.256408	2.499981	17.54373	78.74028	97.37102	100.898	197.1842





For the 8 thread - 500,000 trial run, I think the actual probability of the simulation using the given parameters is 13%. Meaning there is a 13% probability the laser will hit the plate (looking at the results for all 500,000 trial runs there is a definite trend of about 0.13).

To calculate the parallel fraction, I first calculated the speedup from using 1 thread to 8 threads (on the 500,000 trial runs) which is:

Speedup = 197.184204 / 32.750164 = 6.02

Then when considering the 1 thread to 8 thread speedup the parallel fraction is:

Fp = (8 / 7)*(1-(1/6.02)) = 0.95

Explanation:

For this project I used the based code provided with some minor changes to make it work in Visual Studios on a Windows computer. I also added the logic that handles the 4 possible cases by adding if statements that check the variables mentioned in the provided code.

I was having some issues initially when trying to get good runs on the flip server, so I decided to run all my tests locally. In the last project I had tried to use Cygwin64 to simulate a Linux environment but

uncovered that it was not practical. So, I used VS on my Windows machine to create an exe file, I then used a python script to run that exe file. I rewrote the program to accept 3 command line inputs, those being the number of threads, number of tries, and number of trials. I left the number of tries at 10 for all runs. The python script uses two for loops to change the number of threads and number of trials, running the exe for each combination.

My data contains an interesting trend I think may have to do with the initial overhead requirements of generating the multiple threads being used. Looking at the results for 4 and 8 threads, for every trial size other than 500,000 the 4 threads outperform 8 threads. I think only once the number of trials reaches 500,000 (in reality some other number larger than 100,000 and smaller than 500,000) we start to see the gains from having double the thread count. This same behavior can be seen in the graph and tables for every thread size, at a certain trial size the larger thread count begins to outperform all smaller counts. This isn't surprising, I just didn't expect the amount of computations required to make the jump from 4 threads to 8 threads worthwhile. The most vivid representation of this is in the first graph where the number of threads is on the X axis, the decline seen in the lines for the smaller trial sizes shows this decrease in performance as the number of threads is increased. Overall, the data collected follows the expected performance trends described in the lectures.