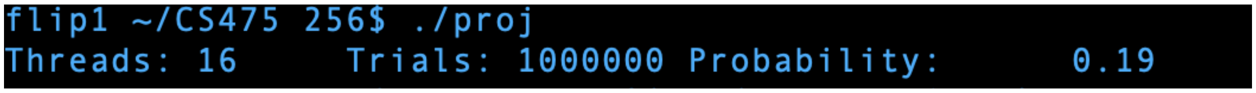
John Olgin
olginj@oregonstate.edu
CS 475 – Spring 2019
Project #1
OpenMp: Monte Carlo Simulation

This project is intended to observe the effects of variable threads and trial runs on performance (measured in mega trials/second) for a Monte Carlo simulation. For our purposes, the simulation is meant to determine the probability of a laser beam making contact with a plate along the x-axis. My program was executed on the flip server at Oregon State. I created a bash file (included in the zip file) that included two loops. The first loop included the thread number set of 1, 2, 4, 6, 8, 10, 12, 14 and 16. The second loop iterated through the set of trials ran, which were 10000, 50000, 100000, 250000, 500000 and 1000000. The range in trial values was intentional and meant to exploit any effects on performance if they existed. My inner loop then compiled and ran my program with using each trial value with the first thread value. When the entire set was used, the outer loop would increment to the next value in the thread number set and the inner loop would execute again. The compiler statement systematically included the changing values as they appeared in their respective loops, giving me 96 overall data points. The bash file inserted each peak performance value into a separate text document for ease of transfer to Microsoft Excel.

My resulting probability from the Monte Carlo simulation was **.19** consistently. This was checked with every combination of trials and threads. It never varied from .19 for any pair of variables. This means the laser beam had a 19% chance of hitting the plate on the x-axis.


```
flip1 ~/CS475 256$ ./proj
Threads: 16     Trials: 1000000 Probability:      0.19
```

Below are the graphs of threads vs. performance, trials vs. performance, and the raw data used to generate the graphs. My program was executed on the flip1 server at Oregon State. It is likely that the inconsistent variations in the data were due to changes of the server load.

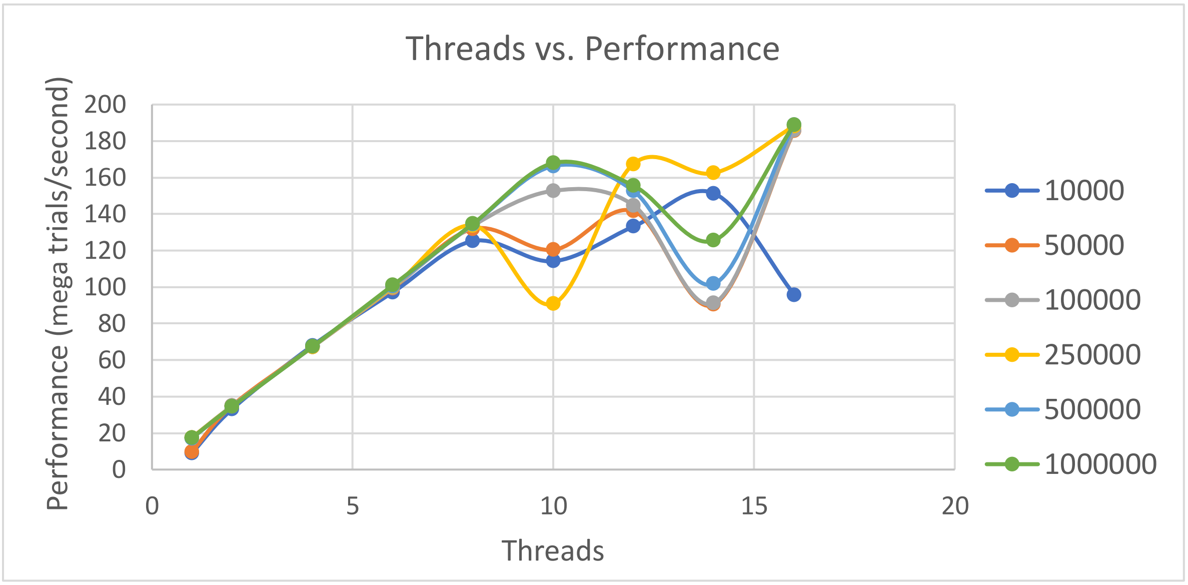
Trials

Threads

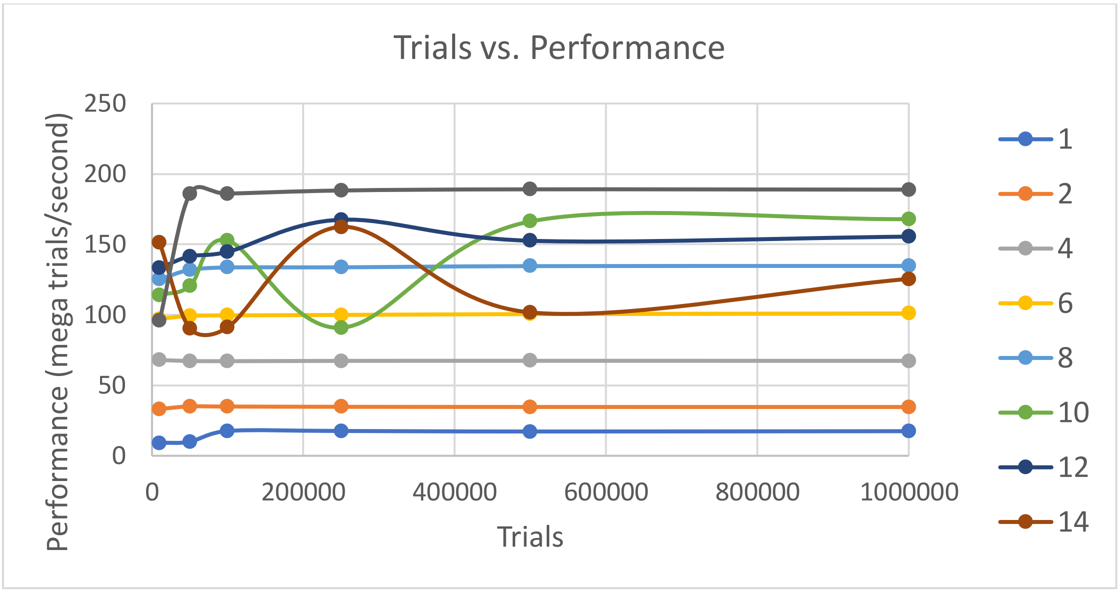
	10000	50000	100000	250000	500000	1000000
1	9.16	9.96	17.62	17.6	17.24	17.47
2	33.21	35.07	34.96	34.77	34.66	34.63
4	67.98	67.27	67.14	67.36	67.41	67.37
6	97.15	99.33	99.63	99.95	100.56	101.01
8	125.22	131.9	133.64	133.76	134.59	134.72
10	114.17	120.39	152.73	90.87	166.31	167.93
12	133.19	141.58	144.71	167.53	152.6	155.53

Performance measured in mega trials/second

14	151.35	90.41	91.2	162.46	101.74	125.57
16	95.87	185.69	186.04	188.26	189.09	188.87



The graph above depicts the effects of variable threads on performance. The general trend is positive, indicating the performance gets faster by adding more threads to the program. Performance became very erratic around the 10-thread mark, which could have something to do with the flip server load at execution time. Increase in performance also seems to taper off after 10 threads. In other words, it becomes less linear. This leads me to expect lower speedup values for the higher thread numbers, specifically between 10 and 16 threads. This could be because the sequential portion is becoming more dominant, thus we are reaping diminishing rewards from each increase in thread count. Judging by this data, I expect that increasing thread volume any more (20, 30, 50, etc.) would result in performance becoming nearly constant to infinity.



The degree of variability in performance based on changes in trial volume is split according to my data. From looking at the graph, we can see that for 1, 2, 4, 6 and 8 threads, the change in performance is insignificant. Performance is relatively consistent despite the large increases in trials ran. However, performance for the higher thread values was very erratic for the lower trial numbers. As mentioned before, server load could have affected the performance for those higher thread numbers. Regardless, even the higher threads seemed to become more constant as we increased trials ran to 250000 and beyond. Additionally, if we take a closer look, we can see the thread counts who have inconsistent performance values roughly coincide with the erratic region (between 10-16) of the first graph.

Speedup
Calculations based on highest trial number (1000000)

S = (mega trials/second for 16 threads) / (mega trials/second for 1 thread)
S = 188.87/17.47
S = 10.81

Parallel Fraction

F_P = (NUMT/(NUMT - 1)) * (1 – (1 / S))
F_p = (16/(16 – 1)) * (1 – (1 / 10.81))
F_p = 1.067 * (1 – (1 / 10.81))
F_p = .968 = **.97**

Alternate F_p calculation (10 threads and 4 threads)

S = (mega trials/second for 10 threads) / (mega trials/second for 4 thread)
S = 167.93/67.37
S = 2.49

F_P = (NUMT/(NUMT - 1)) * (1 – (1 / S))
F_p = (10/(10 – 1)) * (1 – (1 / 2.49))
F_p = 1.11 * (1 – (1 / 2.49))
F_p = .664 = .66

This parallel fraction calculation suggests that a high portion of the Monte Carlo simulation can benefit from parallelism. It also suggests that utilizing 16 threads over 1 thread actually results in doing more work for the simulation rather than setting up to use the additional threads. This is evident by the clear increase in F_p as we move from 10 threads to 16 threads.