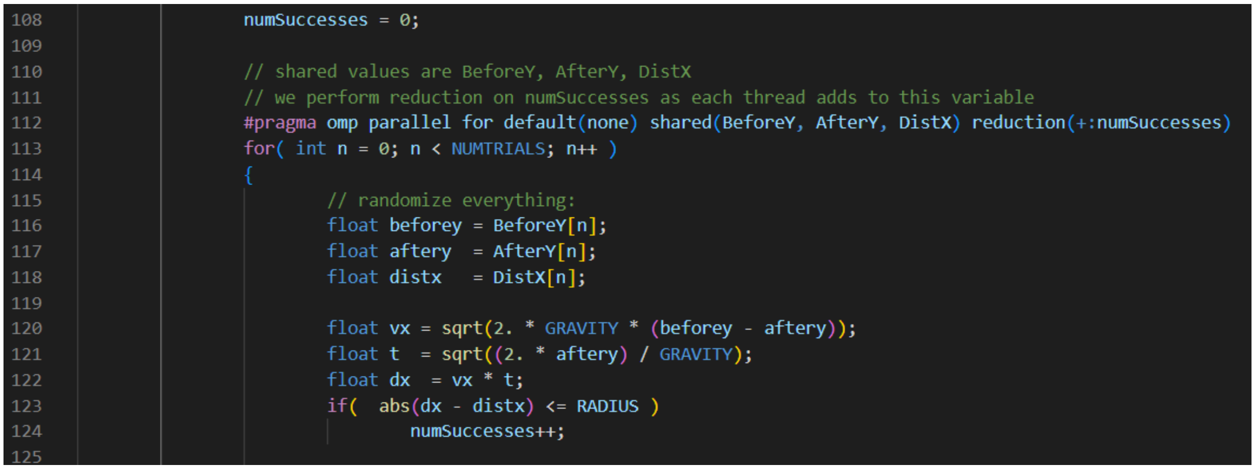
Lyon Kee

keel@oregonstate.edu

Project #1: OpenMP: Monte Carlo Simulation

Key snippets of code



```
108                 numSuccesses = 0;
109
110                 // shared values are BeforeY, AfterY, DistX
111                 // we perform reduction on numSuccesses as each thread adds to this variable
112                 #pragma omp parallel for default(none) shared(BeforeY, AfterY, DistX) reduction(+:numSuccesses)
113                 for( int n = 0; n < NUMTRIALS; n++ )
114                 {
115                         // randomize everything:
116                         float beforey = BeforeY[n];
117                         float aftery  = AfterY[n];
118                         float distx   = DistX[n];
119
120                         float vx = sqrt(2. * GRAVITY * (beforey - aftery));
121                         float t  = sqrt((2. * aftery) / GRAVITY);
122                         float dx  = vx * t;
123                         if(  abs(dx - distx) <= RADIUS )
124                                 numSuccesses++;
125
```

Line 108: We observe that numSuccesses is an integer and not an array where each thread will be incrementing on line 124. We need to handle this on line 112.

Line 112: We are using default(none) thus we will need to specify the variables that are used in the threads. We are sharing BeforeY, AfterY, DistX across threads and we are not altering them so we will just have them as shared to have them all read from these variables. We are writing to numSuccess, so we are using reduction(+:numSuccesses) to update and write to numSuccess in a reduction format which is the fastest followed by atomic and critical.

Tables of data of the performance numbers as a function of NUMT and NUMTRIALS

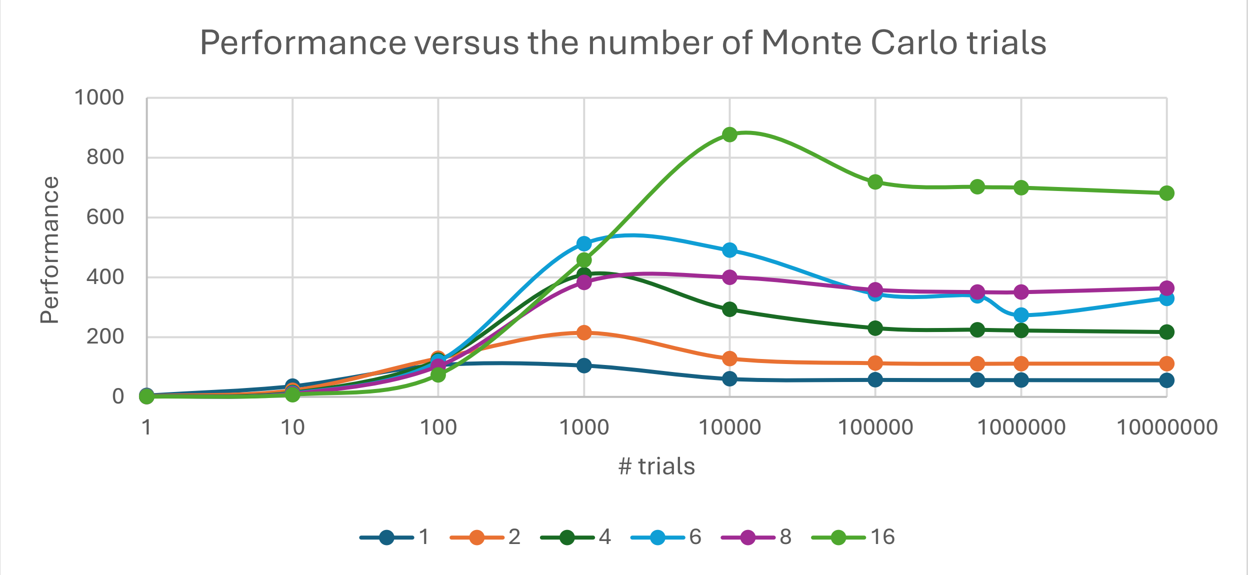
Columns: # Trials

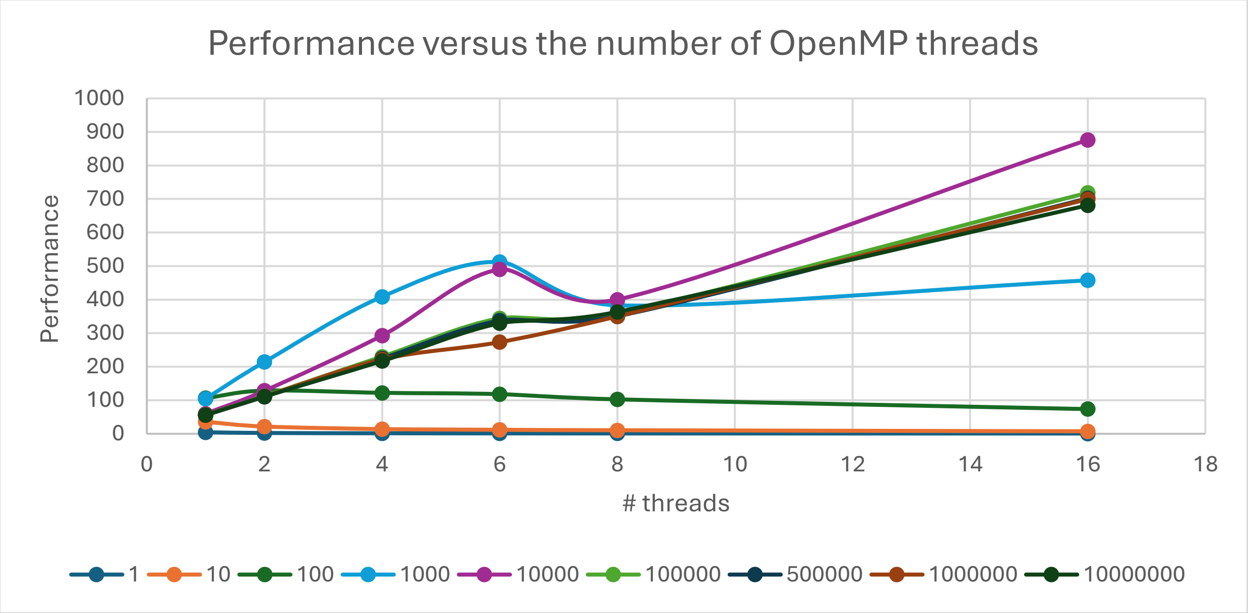
Rows: # Threads

Values: Sum of MegaTrialsPerSecond

	1	10	100	1000	10000	100000	500000	1000000	10000000
1	4.52	35.71	106.16	104.67	59.95	56.76	56.18	55.99	55.49
2	2.42	21.64	128.7	214.27	128.23	112.58	110.61	111.22	111.04
4	1.5	14.04	122.1	408.66	292.84	230.04	224.44	221.94	216.9
6	1.17	11.92	118.2	512.29	490.08	344.31	337.49	273.21	329.52
8	0.99	10.49	102.67	383	400.08	357.88	350.38	349.99	363.34
16	0.73	7.47	73.64	457.46	876.42	718.77	701.91	699.55	681.41

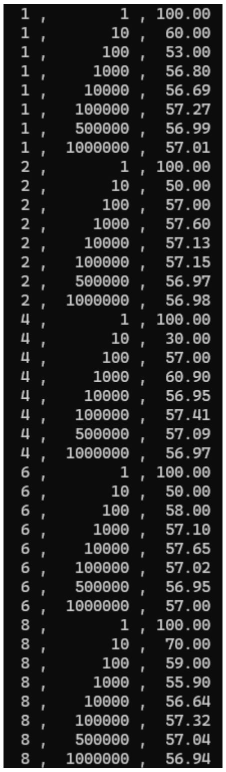
Graphs of data





Conclusion

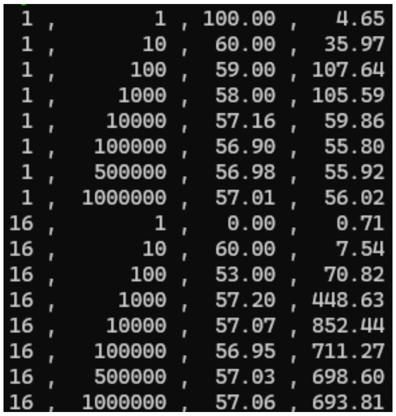
1. Your estimate of the Probability.



```
 1 ,       1 , 100.00
 1 ,      10 ,  60.00
 1 ,     100 ,  53.00
 1 ,    1000 ,  56.80
 1 ,   10000 ,  56.69
 1 ,  100000 ,  57.27
 1 ,  500000 ,  56.99
 1 , 1000000 ,  57.01
 2 ,       1 , 100.00
 2 ,      10 ,  50.00
 2 ,     100 ,  57.00
 2 ,    1000 ,  57.60
 2 ,   10000 ,  57.13
 2 ,  100000 ,  57.15
 2 ,  500000 ,  56.97
 2 , 1000000 ,  56.98
 4 ,       1 , 100.00
 4 ,      10 ,  30.00
 4 ,     100 ,  57.00
 4 ,    1000 ,  60.90
 4 ,   10000 ,  56.95
 4 ,  100000 ,  57.41
 4 ,  500000 ,  57.09
 4 , 1000000 ,  56.97
 6 ,       1 , 100.00
 6 ,      10 ,  50.00
 6 ,     100 ,  58.00
 6 ,    1000 ,  57.10
 6 ,   10000 ,  57.65
 6 ,  100000 ,  57.02
 6 ,  500000 ,  56.95
 6 , 1000000 ,  57.00
 8 ,       1 , 100.00
 8 ,      10 ,  70.00
 8 ,     100 ,  59.00
 8 ,    1000 ,  55.90
 8 ,   10000 ,  56.64
 8 ,  100000 ,  57.32
 8 ,  500000 ,  57.04
 8 , 1000000 ,  56.94
```

This is a screenshot of the run and we observe that the values converge to an average of 57% when the number of trials are high.

2. Your estimate of the Parallel Fraction



```
 1 ,       1 , 100.00 ,   4.65
 1 ,      10 ,  60.00 ,  35.97
 1 ,     100 ,  59.00 , 107.64
 1 ,    1000 ,  58.00 , 105.59
 1 ,   10000 ,  57.16 ,  59.86
 1 ,  100000 ,  56.90 ,  55.80
 1 ,  500000 ,  56.98 ,  55.92
 1 , 1000000 ,  57.01 ,  56.02
16 ,       1 ,   0.00 ,   0.71
16 ,      10 ,  60.00 ,   7.54
16 ,     100 ,  53.00 ,  70.82
16 ,    1000 ,  57.20 , 448.63
16 ,   10000 ,  57.07 , 852.44
16 ,  100000 ,  56.95 , 711.27
16 ,  500000 ,  57.03 , 698.60
16 , 1000000 ,  57.06 , 693.81
```

Speedup, S = 852.44 / 59.86

= 14.24

Parallel Fraction, F = n/(n-1) * (S - 1)/S = 16/(16-1) * (14.24 - 1)/ 14.24

= 0.992

This is to be expected because we have a high number of trials which will lead to a higher parallelizable fraction according to Gustafson-Baris Observation. When n = 100, parallelizable fraction is 0.36, following the working above but with values of trials = 100

3. Graph commentary

Looking at the "performance vs the number of monte Carlo trials" graph, we observe that when we set 16 threads, we obtain the highest number of performance, this is in accordance to a system with 16 threads. During low number of trials, we observe that there is not much difference between the number of threads and the performance it gives. Only after about 1000 trials, we see a split in performance where the higher number of threads would come out on top on performance. At 1000 trials, we observe that 16 threads didn't come out on top, this is because the number of trials is low and having 16 threads running together would just slow down things due to overhead cost of scheduling. At about 10,000 trials, we observe a peak in performance, this is likely due to system limitations where a higher number of trials, did not lead to more performance even though it promotes a higher parallelizable fraction at 10,000,000.

In the graph "Performance versus the number of OpenMP threads", we see a constant upwards trend, as the number of threads is on the x-axis, we see that performance increases as the number of threads go up. When we look at the different number of lines that are graphed, we see that the highest number of trials did not yield the highest performance, this is again due to system limitations as mentioned above even though the code promotes a higher penalizable fraction, but the jump from 100,000 to 10,000,000 is minimal as compared to the jump from 1 to 10,000. From this graph, we observe that the optimal number of trials for performance on my system is 10,000 due to memory behaviors.

NOTE: The program is ran on my personal laptop so there are a lot of background threads running even during idle.

How did things turn out?

Things turned out well with a weird peak at 10,000 trials. Other than this, the "Performance versus the number of monte Carlo trials" graph initially turned out weird with hard to read graphs. We resolved this issue by making the x axis a logarithmic scale, this allowed us to see values at the lower end of trails as it scales up from 1 to 10,000,000, which then tells us a lot more information about how parallelization performs.

Why do you think they turned out that way?

The peak at 10,000 runs is likely due to memory issues and possibly due to caching as the number of trials increases, memory usage goes up and performance will decrease due to the need to access main memory to retrieve value instead of using caching.