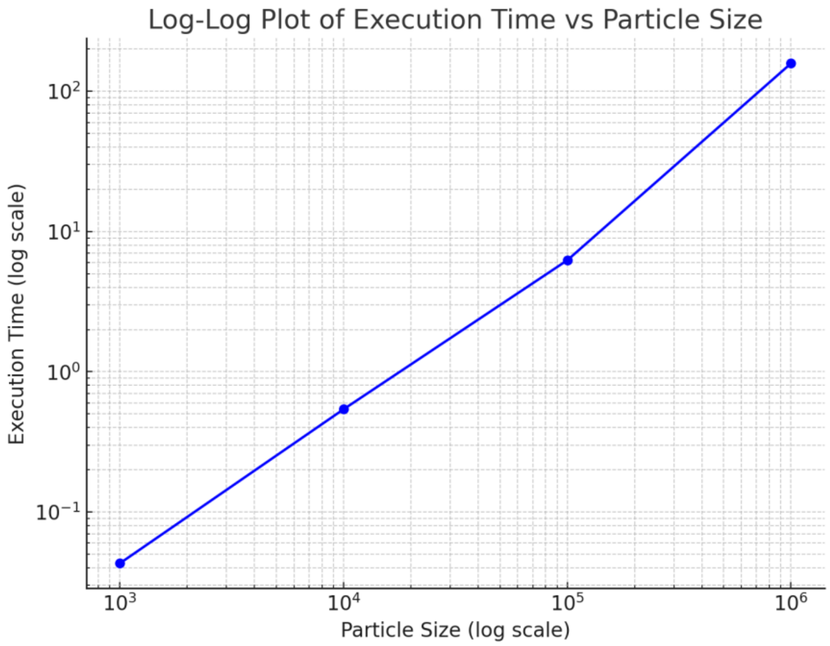
HW2: Shared Memory Particle Simulation

Lian Liao, ll987, ll987(Perlmutter)

Serial code:



The plot in log-log scale for serial code.

I noticed that the performance is a little bit different between submitting the job and running the code in the node directly. However, I picked the worst one, which is submitting the job.

The result is very straight. It is almost perfect below the 10e^5 size. The LSC is 1.1759. Although it is not that low, my execution time performance is relatively good. The 1e^6 size only costs 158 seconds which is my Runtime Serial, much lower than recitation.

<u>Final code description:</u>
For the approach. I didn't change any of apply_force and move function. I simply initiate a 3D vector grid, essentially a 2D grid where each cell contains a list of particles. The gridCellSize = cutoff and gridSize = particle size / gridCellSize. I define them and fill the grid in init.

For the simulate_one_step, The grid is traversed, and forces are calculated between particles within the same cell and in adjacent cells. Call the apply_force as well. In the last move the particle, but If a particle moves to a different cell after being moved, it is removed from its current cell and added to the new cell.

My serial code wasn't like this; I optimized it when I implemented the OpenMP code, then copied the serial version to be serial code. I will explain the technique used for the serial part here.

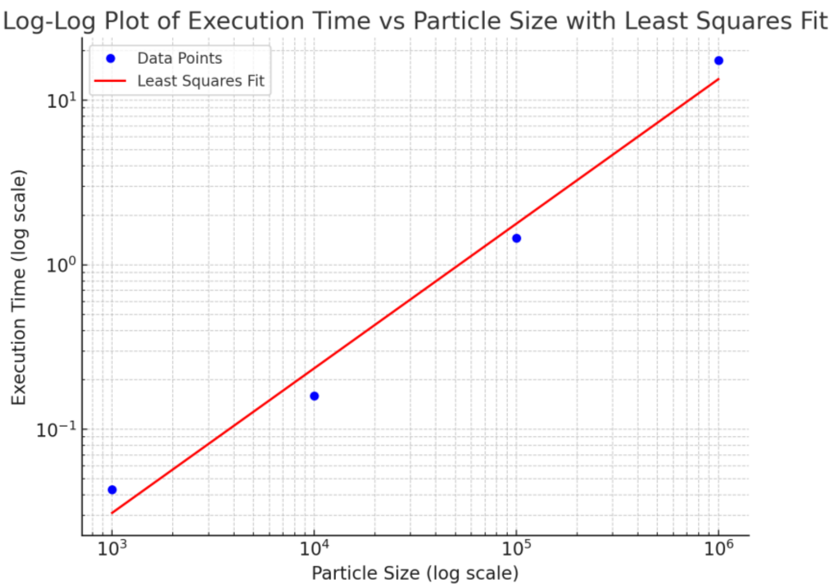
Description of the design:

The most different is that my original implement is clear the grid and fill it in every step. This is a waste of too many resources, so I improved it and got a lot of increased performance. Another thing is I combine the for loops. In the end, I only have two main loops: one is compute, and the other one is move. Also, the loop seems a mass because I had some functions to be clear, but I decided to put them all in the simulate_one_step because it is clearer for me to implement the open MP. Once you have the grid implemented, it can reach the O(n), I believe.

Potential Improvement:

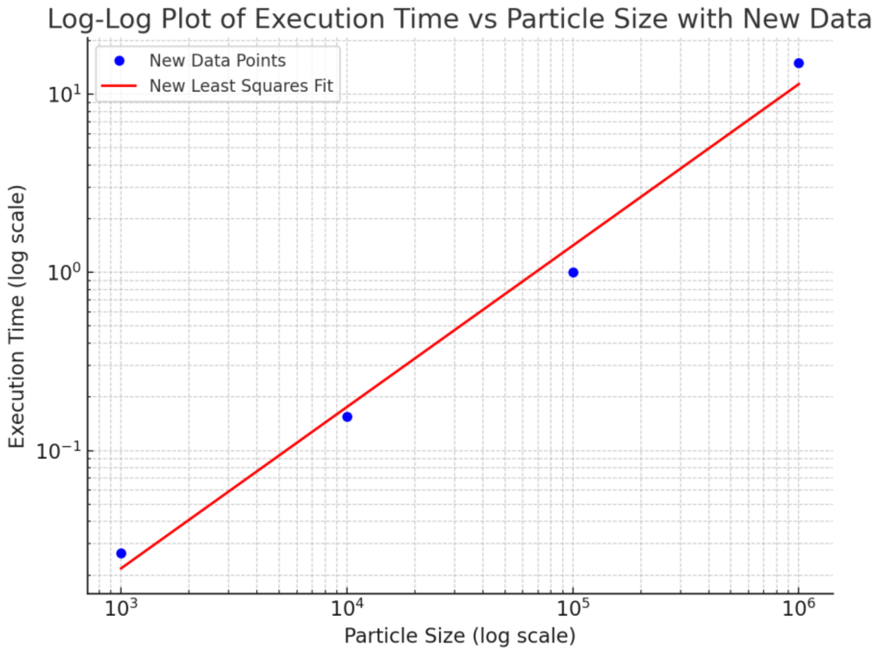
I think my logic is correct, but maybe the grid structure can be better, like using only a 1D vector. I didn't touch the apply_force and move function. Maybe they can have more efficacy.

OpenMP code:



The plot in log-log scale for 16 threads

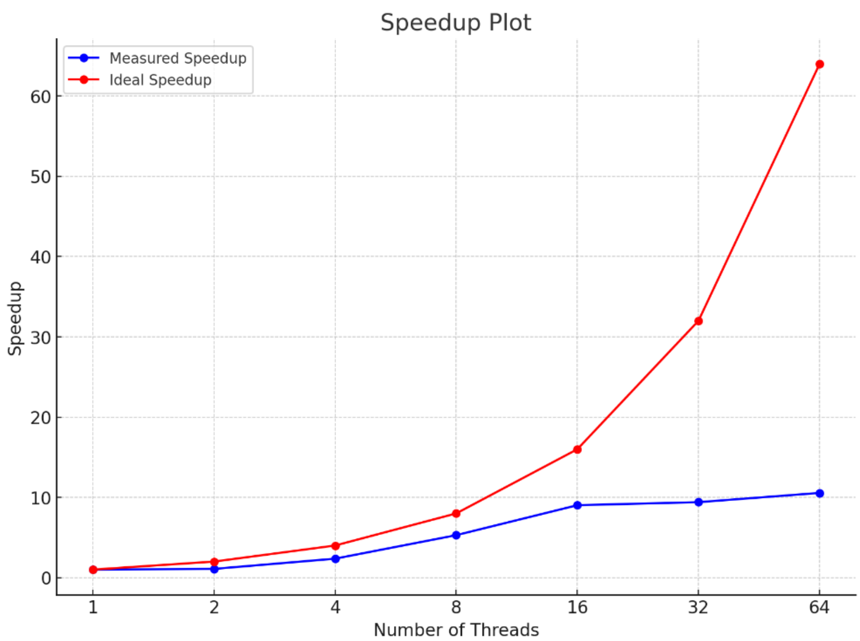
Efficiency(strong)=0.564



The plot in log-log scale for 64 threads

Efficiency(strong)=0.165

The result of 16 and 64 threads are pretty similar. They show the good performance of O(n).



The speedup plot from 1 to 64 threads

It is very similar to the recitation result. The curve is flat after 16 threads.

Final code description:

Only three stuffs add to serial code for parallel.

1. #pragma omp for collapse(2) for the compute loop.
2. #pragma omp for for the move loop
3. The lock to protect grid data

Description of the design:

As I mentioned. I combine many loops into two main loops, so my OpenMP code seems very simple. I have tried to add other clauses. I have tested so many combinations. The result is the best if keep it simple. One clause I want to point out is schedule(), regardless of whether it is dynamic or static. It would slow down the speed a lot. I thought it was suitable to handle complicated computing; maybe it is not very complicated.

For the lock. There are some choices: barrier, critical, atomic and lock. I have tried all of them. It didn't affect speed that much between those. But here is what I feel:

Lock> atomic> barrier> critical.

They are all to avoid data conflict. Very important.

Potential Improvement:

Because it is in the parallel region initially, separating the gird as private and combining them in the end is hard. I believe it is worth trying.

I also tried to separate the job to each thread, but it would slow down the speed, so I undo it in the end. However, I think it is still a potential wat to improve.

Unexpected behavior and comparison with a different machine:

Odd behavior:

For the OpenMP code. I got so many memory errors in the beginning. It is very hard to debug if I just copy the serial code into it.

Due to my wrong implementation, setting more threads is slower. I think it is because I set an inner parallel region, which is not right. Also, the schedule()! Trap.

The result is different between submitting the job and running it directly. I guess submitting the job has some other tiny addition process?

I can set up the threads more than actually have! I didn’t know that. there are software threads and hardware threads.

<u>Different machine:</u>

It is slower on my computer on average, but the LSC are pretty similar, which is in expectation. However, on my computer. The run time is always dynamic. I guess it is because my computer has many other processing jobs. Overall, the plot is similar, just slower and more unstable than the supercomputer.

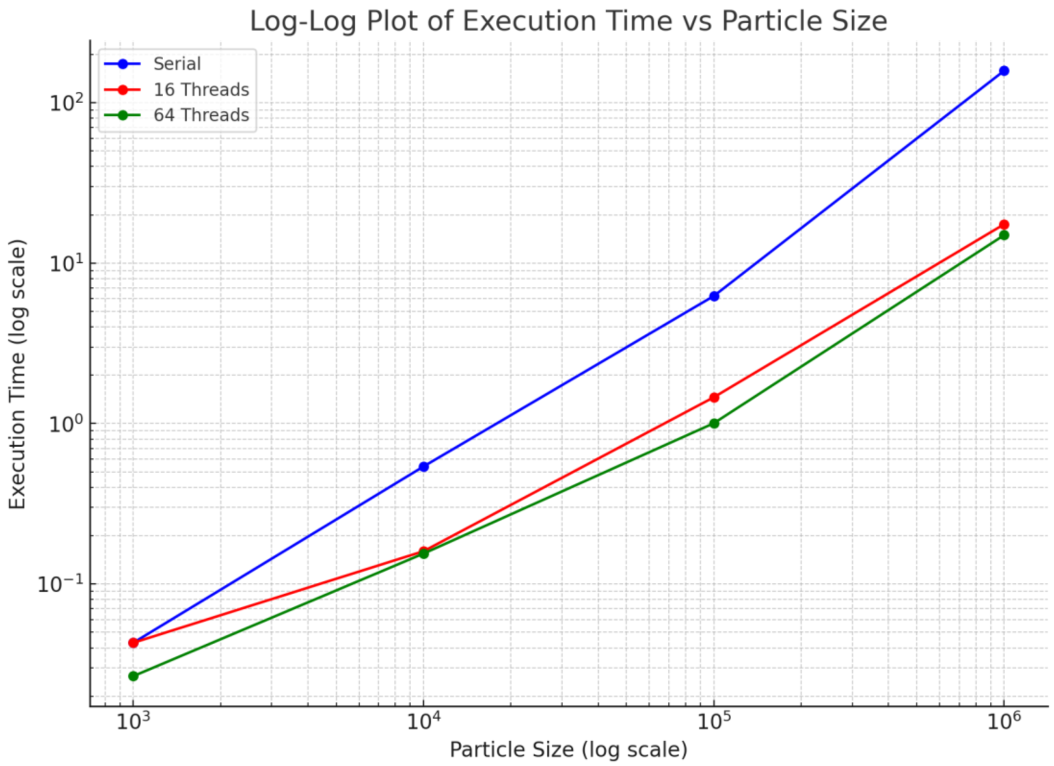
Note:

I ran all the code with seed 1

The plot drawn by AI

My calculation of the total grade is 1.15

Append:



Comparison of 3 different threads in scale

$$\mathrm{Speedup}(p) = \frac{t(n,1)}{t(n,p)}$$

$$\mathrm{Efficiency}_{\mathrm{strong}}(p) = \frac{\mathrm{Speedup(p)}}{p}$$

Formula of efficiency(strong)