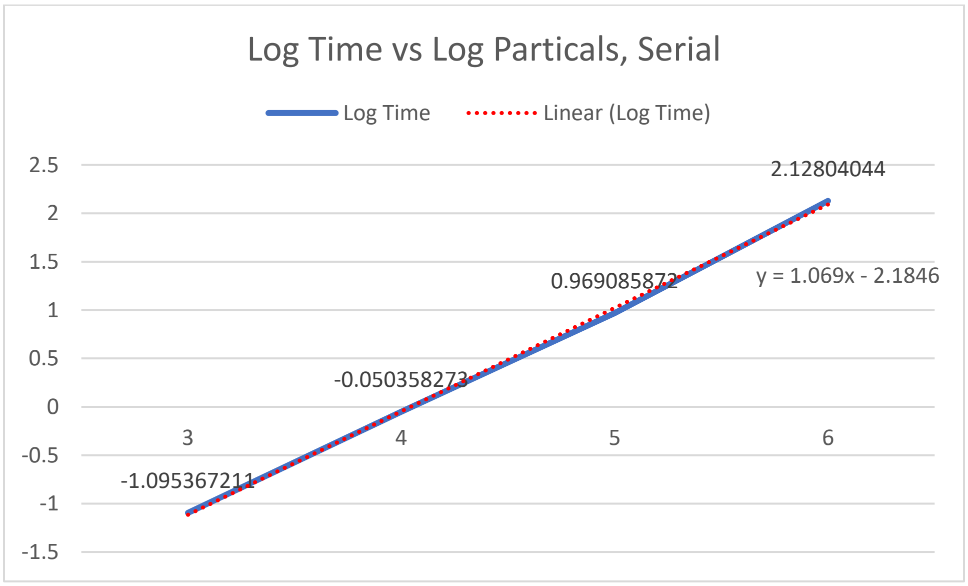
CS 5220 HW2 Report

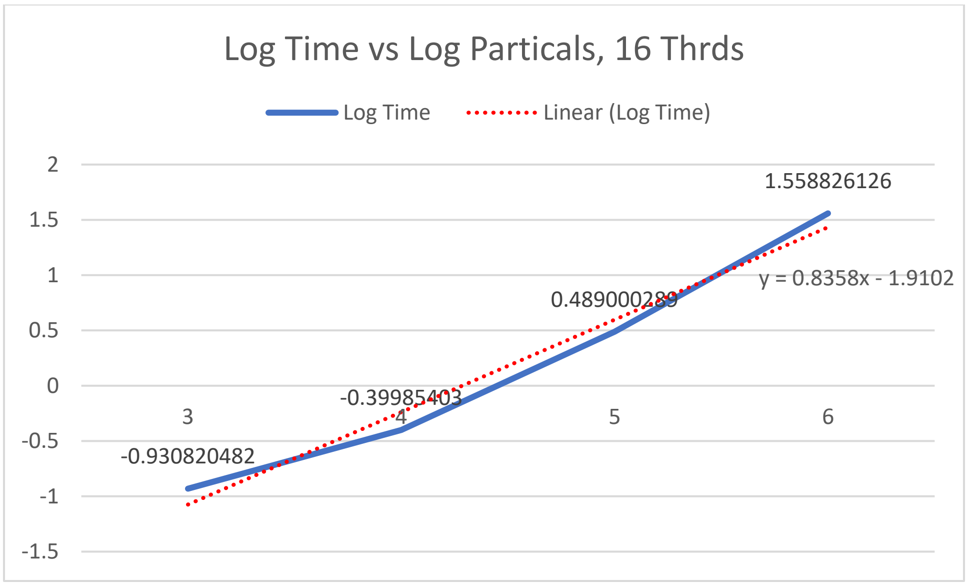
Name: Santiago Lai, Cornell id: zl345, NERSC username: Zhengnan

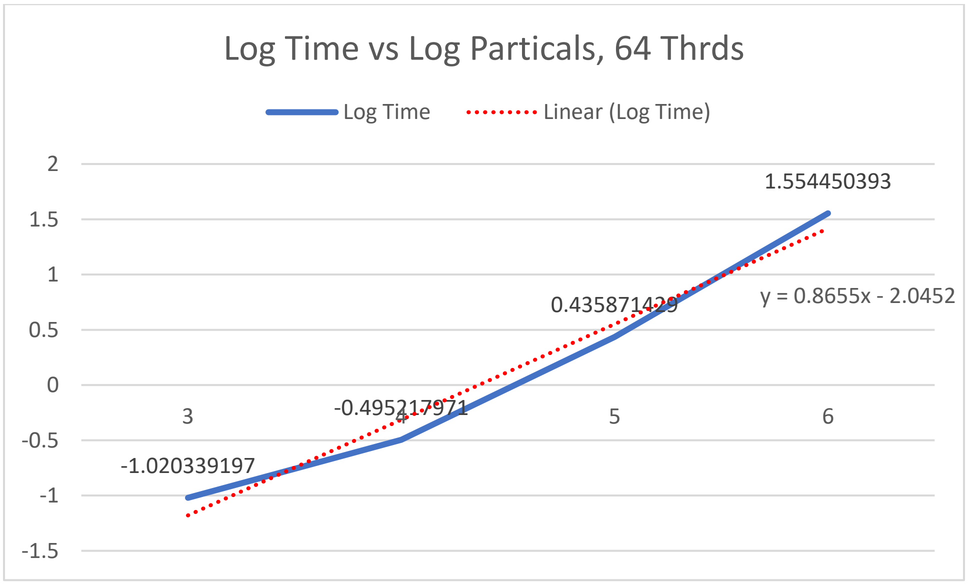
In this report, I will discuss each of the optimizations I attempted in serial and shared memory implementation, from data structures and synchronization I used to some design choices in my code. Lastly, I will describe my OpenMP code performance vs the idealized speedup, in addition to a breakdown of runtime into computation time/ synchronization time and communication time.

0. Plot in log-log scale for my serial and parallel codes:

The three plots for runtime by serial, 16-thread parallel, and 64-thread parallel codes are shown below. In a log-log scale, all the plots achieved a slope of near 1 (1.067, 0.8358, and 0.8655, respectively). The results show that my implementation successfully improved the performance from quadratic runtime to linear runtime.







1. Data structures in serial implementation

In my serial implementation, I followed the instructions in the handout to perform particle binning. I used a three-dimensional array to store the particles, where the first dimension is the x-axis of the bins, the second dimension is the y-axis of the bins, and the third dimension is the bins that store the particles. To speed up my implementation, I did not use the vector implementation provided by C++, since it's on heap and is relatively slow.

Instead, I wrote my own dynamically allocated array, which doubles the size if there are too many particles in one bin. Such data structure is advantageous in two perspectives: it is relatively fast as it's on stack, and it would not use too much memory.

2. Design choices in serial implementation

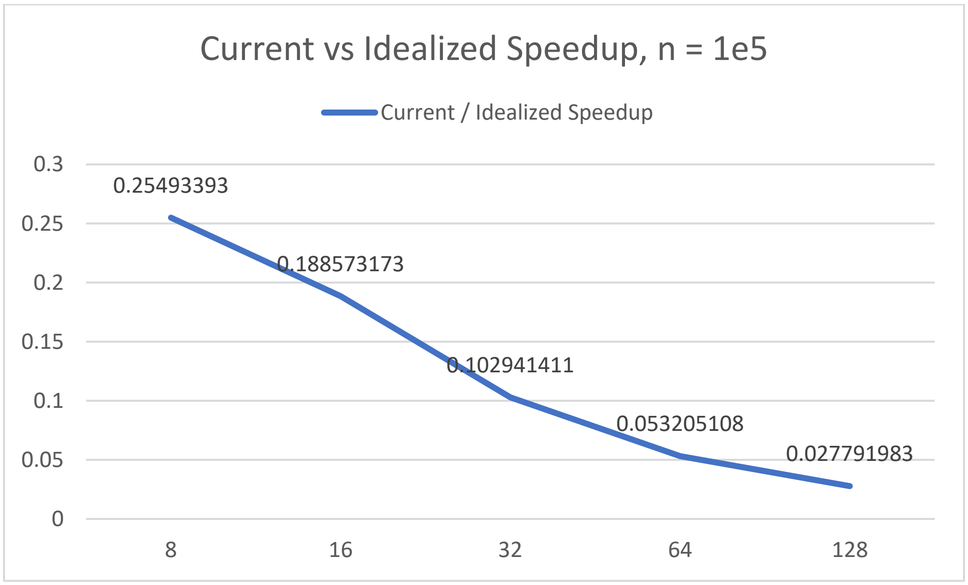
One of the most important parts in my serial code implementation is the size of bins. I hope to choose a bin size so that I'll only need to visit the 8 nearby bins when applying force, thus reducing the runtime complexity from quadratic to linear. Considering edge cases where a particle is at the corner of a bin, the bin size should be at least the cut-off size. However, I later noticed that there is a tradeoff between large and small bin sizes. If the bin size is small, there will be less particles in each bin, meaning that we'll consider less interactions. However, there will be more bins that require reinitialization at each step, thus increasing the runtime. By experimentally setting the bin size to different multiples of cut-off distance, I found that the runtime is the best when the size is set to 5 times the cut off distance. I kept using this value in the rest of my serial and parallel implementations.

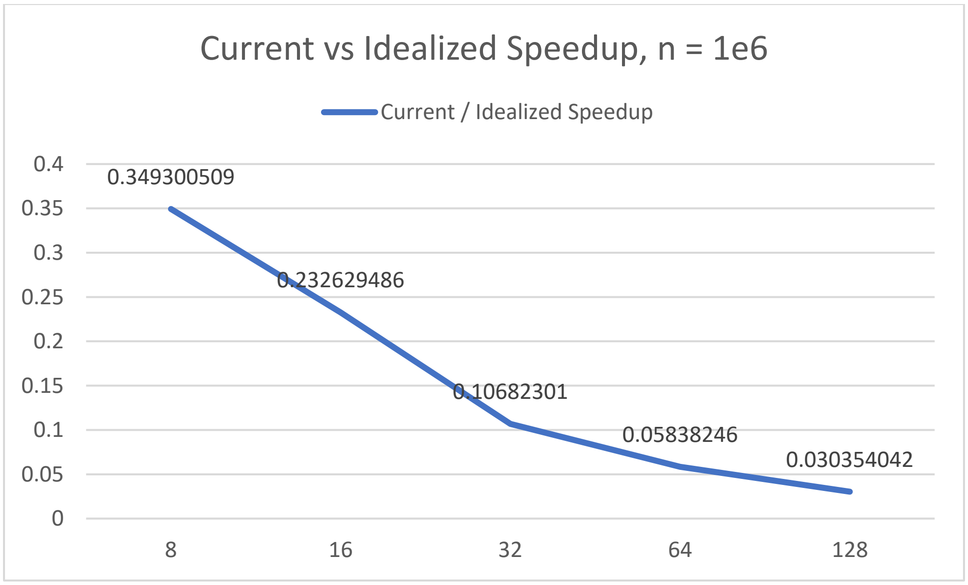
3. Synchronization in parallel implementation

In my serial implementation, there are mainly 4 procedures to simulate one step: reinitialize the bins, put the particles into bins, compute forces, and move particles. Every step can be done parallelly except putting particles into bins, where a race condition might happen if 2 particles are put into the same bin parallelly. Therefore, I simply used #pragma omp for to synchronize the other three steps, whereas for particle binning I created an array of omp_lock_t to make sure no two particles can visit the same bin at the same time. It was slightly faster than simply using #pragma omp simple that I tried, which executes the procedure with one single thread.

4. Speedup by parallel implementation

Please see the plots for the speedup by my parallel code with different number of particles.

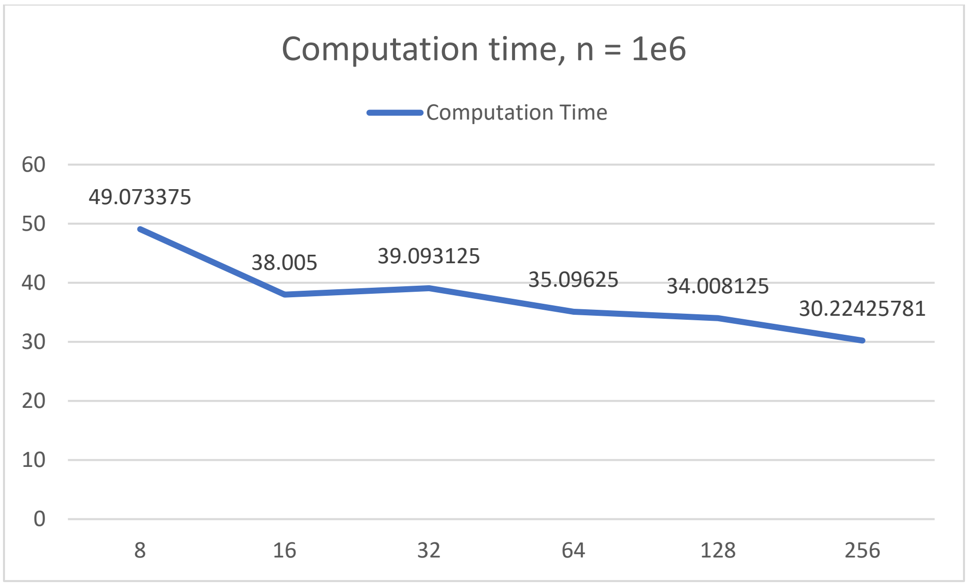


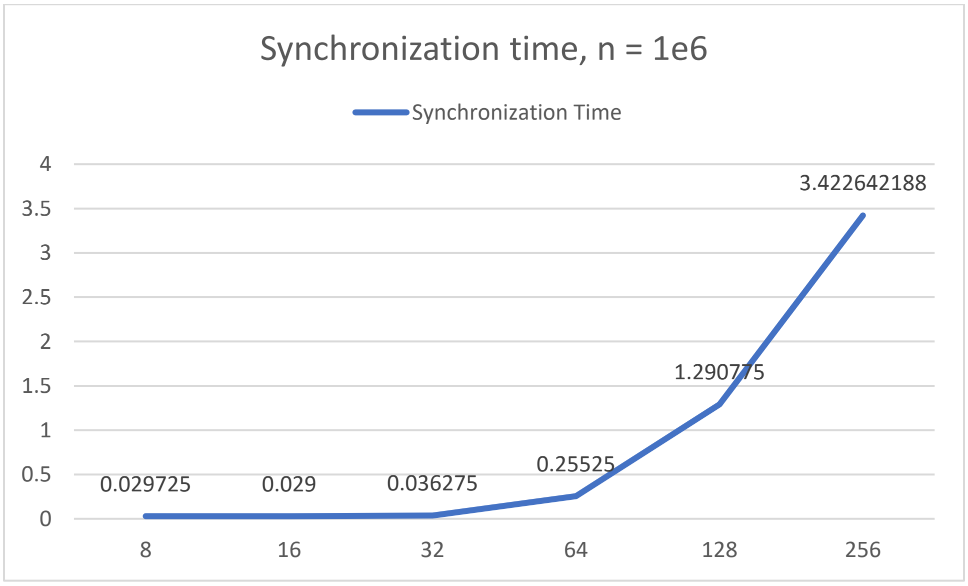


It is hard to do better because there's a significant portion of my code that is non-parallelizable, i.e., putting the particles into bins. It implies that my parallel efficiency will approach zero as the number of processors increases.

5. Runtime decomposition for parallel implementation

I plotted the runtime breakdown for 1000000 particles with number of processors being 8, 16, 32, 64, 128, 256.





We see that the computation time decreases slightly as the number of processors increases, whereas the synchronization time increases significantly if we increase the number of processors. This is consistent with our intuition.

6. Unexpected Behaviors

In my implementation, there were mainly two unexpected behaviors, one in serial code and one in parallel code.

In serial code when considering the appropriate bin size, I once computed that the number of bins N = $\sqrt{M}$, where M is the number of particles, since resetting and inserting into bins takes O(N^2) time and applying forces takes O(N^2/M^2) time. It turned out that the performance was much worse. The reason might be that inserting particles into bins was not computationally heavy compared to applying forces.

In OpenMP code, before using locks for inserting into bins I simply used a #pragma omp single. I expected a significant improvement after changing to locks, however it wasn't. Again, the reason might be that inserting particles into bins was not computationally heavy, thus replacing it with locks did not have noticeable improvement.

7. Conclusion

After implementing particle binning, my code efficiency increased significantly from quadratic runtime to almost linear runtime. By parallelizing the serial code with OpenMP, the performance did improve by limited extent, which is consistent with the Amdahl's Law.