Sahil Patil

Ursan Tchouteng Nijike

CS267

24 February 2025

# Homework 2 Write-Up

*Introduction*

Particle simulations are a very important part of chemistry, as well as computer science. Being able to optimize these simulations and make them as efficient as possible is crucial to studying fluid dynamics, materials science, and physics at a very small scale. This brings us to the main problem of this assignment- optimize a serial O(n^2) particle solution so it is O(n), then use OpenMP to parallelize the O(n) code.

*Contributors*

**Ursan** - Debugged the OpenMP simulation & got it working. Also tested out all simulations so they run properly without issues. Also contributed to the write-up.

**Sahil** - Got serial simulation to O(n) & contributed to the OpenMP simulation. Also contributed to the write-up.

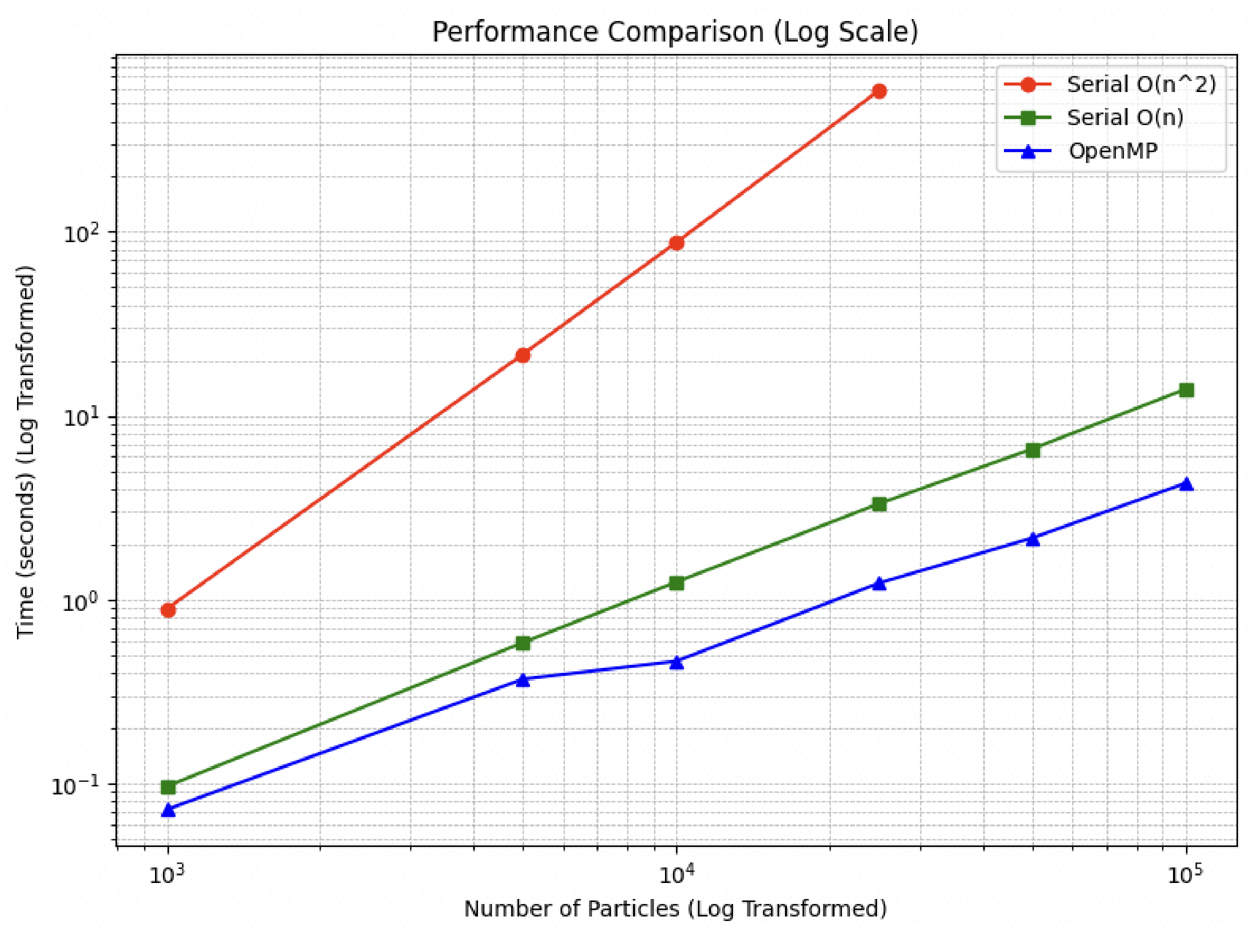
**Vint** - Dropped class. Was not part of this assignment.

*Optimization Techniques Used*

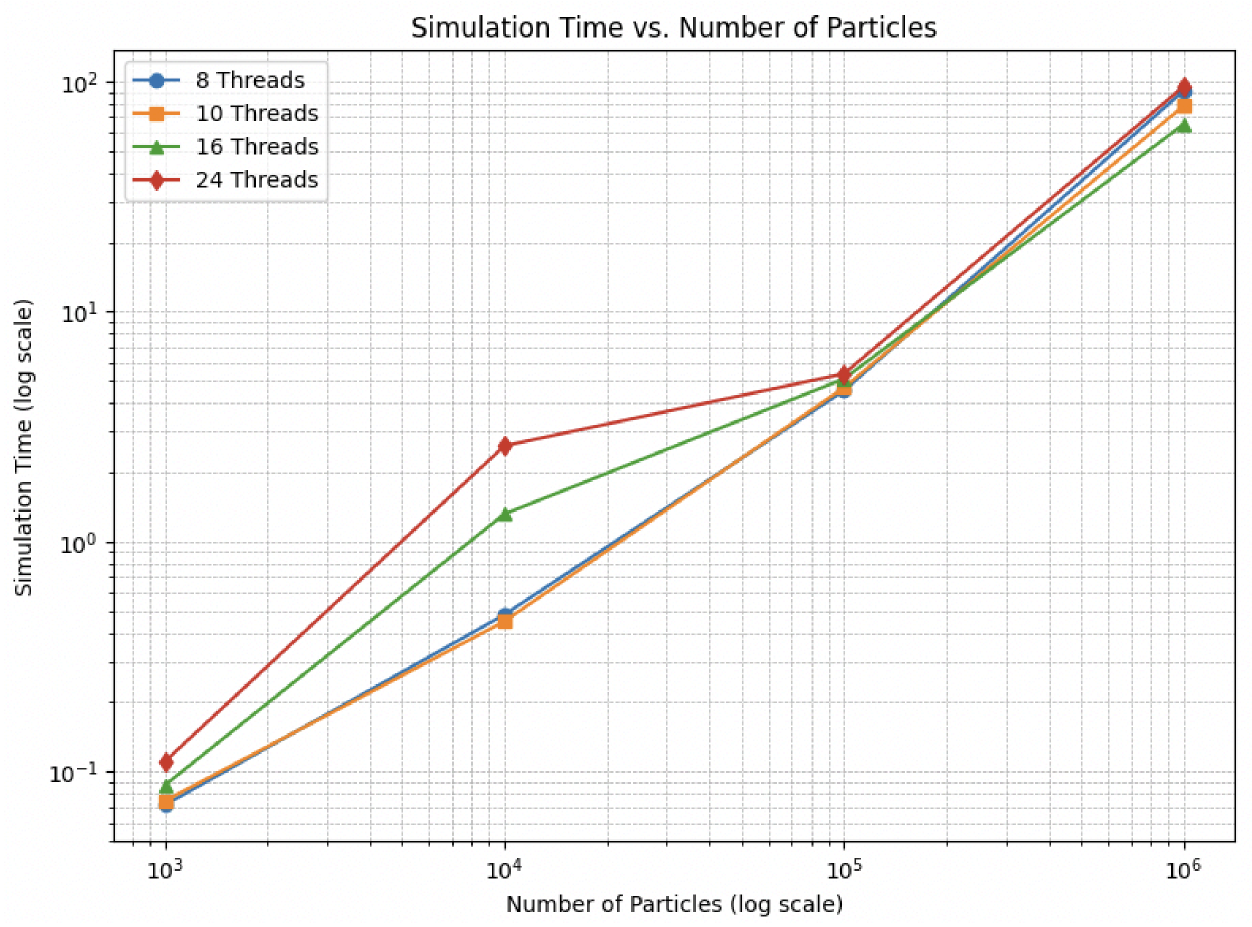
To start things off, the serial O(n^2) was brought to O(n). This was done by using spatial partitioning to limit interactions to particles that are within a cutoff distance. A grid of bins was created. The particles were then assigned to a bin. Force calculations were only done for molecules close. By creating a bin and setting a cutoff distance, there is no need to check parameters between a molecule and every other molecule. This modification brought the run time from 1.17 seconds for the O(n^2) to 0.05 for the O(n).

After this, the next step was to parallelize the O(n) implementation so it now uses OpenMP. This presented many challenges as the code kept crashing due to segmentation faults. After carefully inspecting the entire simulation program, we were able to better understand how OpenMP was applied to the openmp.cpp file. It was clear that <#pragma omp parallel> was active when running the openmp.cpp file even without any openmp command within the file. Armed with that knowledge, the first step was to identify needed barriers or critical sections to our serialized implementation. Our first approach was to leverage implicit wait of the <omp for> command. As a result, we added two <omp for> command to our code, which resulted in a performance of around 1.6 s on 1000 particles. This code implementation however performed better than the unmodified serialized one due to the O(n) computational complexity benefit of our modification. This meant that the poor performance was mostly due to the overhead present in the creation of threads, which exceeded the benefit of applying parallelism. Our second optimization of the parallel code consistent of replacing the implicit barrier of the <omp for> loop with explicit <omp barrier> at the appropriate locations. This also did not result in a significant improvement.  Our final implementation consisted of implementing 1 explicit <omp barrier> and  3 <omp for> with nowait instruction as well as introducing some local variables to thread in an attempt to prevent false sharing. We observed 100000 particles complete the simulation in just 6.6 s when our O(n) optimized serialized model completed in 15 s.

*Results of Optimization*



The above plot was created by setting the number of threads to 10. Unsurprisingly, the serial (un-optimized) version of the simulation was the absolute slowest. The optimized serial showed great improvement for all numbers of particles and the OpenMP was the best amongst the three options. The OpenMP simulation was similar to the performance levels of the optimized serial simulation when the number of particles were lower. But as we crank up the amount of particles, the performance of the OpenMP simulation sees a significant divergence from the optimized serial. As mentioned above, the serial was optimized by using a grid that stored bins. Within these bins were the particles in question. These bins made sure that the program wouldn't have to compute forces between a molecule and all others in the simulation. The results visualized above do not surprise our group; OpenMP is supposed to increase the



performance of a program when implemented properly.

Looking at the second graph, results begin to puzzle us. When the number of particles were kept low, the simulations ran at 8 and 10 threads performed better than the ones ran on 16 and 24 threads. As the number of particles increased, all thread configurations performed almost the same; however, it does appear that the 16 thread simulation performed the best amongst the pack- albeit at a very negligible level. For our specific program, assigning more than 16 threads seems to give diminishing returns when it comes to runtime. Beyond 16 threads, it seems that the overhead from thread management (context switching, memory access, load imbalance, etc.) is hampering the overall performance.

## *Conclusion*

Parallelizing a particle simulation has presented many challenges, but also many learning opportunities. This exercise has certainly made me more appreciative of high-performance computing and the work needed to get a program to run at its theoretical peak. It was far from easy, but it was absolutely rewarding. Getting the OpenMP implementation to finally work was well worth the effort needed to get it up and running. The performance of the OpenMP simulation did not match the theoretical peak and future work on this project could involve dealing with and fixing the false sharing issues that our group ran into when setting a high number of threads.

*References*

1. TylerMSFT, “OpenMP functions,” Microsoft Learn. [Online]. From: https://learn.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-functions?view=msvc-170.