CS5220: HW2 Optimizing Particle Simulation

Name: Andrew Cheng, Cornell ID: 5128767,

Cornell netid: akc55, Perlmutter username: andyhero

CS 5220 Spring 2024

1 Introduction

The core of this assignment is optimizing particle simulations. For this assignment, we have to optimize the total simulation time of particles. In the naive version of the execution, each step would scale quadratically $O(n^2)$ because you would need to calculate the impact of each particle on any of its possible neighbors. The simulation moves over 1000 steps, and we need to minimize the time. Due to the quadratic nature of the problem, there are many ways to optimize the algorithm's execution. These optimizations include binning and many cache optimizations for optimal execution.

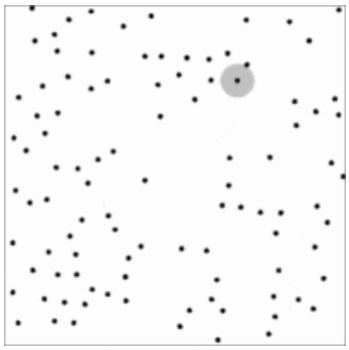

Figure 1: Final Peak Performance With Block Size 4

2 Personal Performance

For personal performance within this homework, I achieved a serial runtime of 80 seconds with an LSC of 1.11. My parallel execution of 16 threads was 19 seconds, and my parallel execution of 64 threads was 12 seconds. Overall, the main performance improvement I created was the implementation of efficient binning. In the first way I binned my data, I still looped through all the surrounding particles within the eight nearby bins. However, I noticed that since the force is applied in equal and opposite directions, you only need to apply this dual force on the bins that you have visited. This means that you only need to apply the force on the bin to the right and the three bins right below this bin to the right and the left. This reduces the total number of function calls to apply force down by half and leads to a better cache structure because there are only a few bins that actually need to be updated. The second most effective optimization was the inlining of every function; because there is only one file for the HW, the code size isn't that big of a concern for the application. Furthermore, inlining provides the compiler with better insight into how to incorporate the function in the overall compilation unit. The introduction of inlining improved the overall serial performance by around 30 seconds. Each of these optimizations comes together to generate better and faster code.

3 Time Spent Within Algorithm

There is a distinct distinction between the parallel and serial execution time ratios between computation and binning. Within serial, the two scales roughly evenly as the number of particles increases. However, in parallel, the rebinning step scales much worse than the computation step of the algorithm. This can be mainly attributed to the contention caused between threads. Within the computation step, the parallel code only needs to create a lightweight atomic update for each particle. This results in limited contention between threads as looping over the bins between threads leads to limited update contention as the particles will be further away from each other because they are not iterating over bins next to each other. Within the serial component, the amount of total execution doesn't change. This results in the ratio between bins and particles to remain roughly the same. Furthermore, the natural iteration pattern of going through particles or bins leads to better cache results of the total execution. Within the parallel execution, there may be false sharing and difficulties in syncing data between the threads.

For a focus on rebinning, the parallel execution has an increased amount of contention because putting the values back into nearby bins requires me to lock the bins before accessing them. Thus, the parallel scaling ratio is much worse for the rebinning step than the computation step. The serial approach doesn't have this issue because all the memory it is accessing is contained within its own main thread. I don't believe there is much communication time within the algorithm, as the memory is all shared between the threads. The main concern for the parallel execution is the cost of synchronizing the bins between each particle update.

For the calculation of these values, I utilized the built-in Chrono library as well as a few aggregators within my C++ files. Within the files, I would calculate the hardware time on the master thread prior to the execution of a single step. After the single step of computation terminated, I would write down the time for computation. I follow the same procedure for rebinning the particles, as seen in Figure 2.

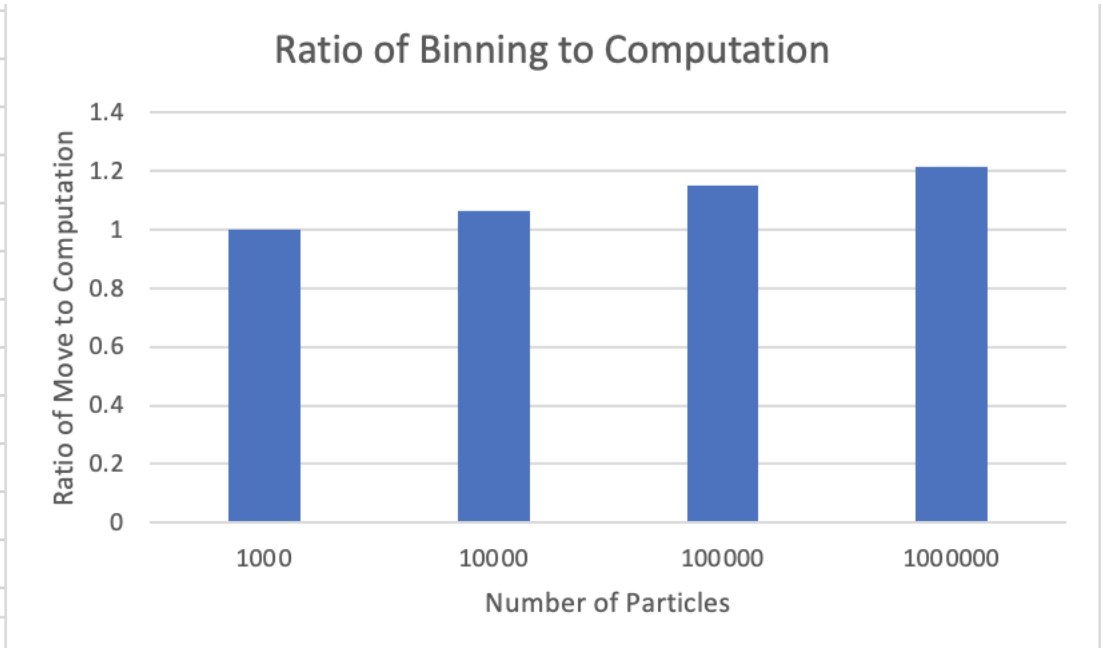

Figure 2: Final Peak Performance With Block Size 4

4 Data Structure Design Choices

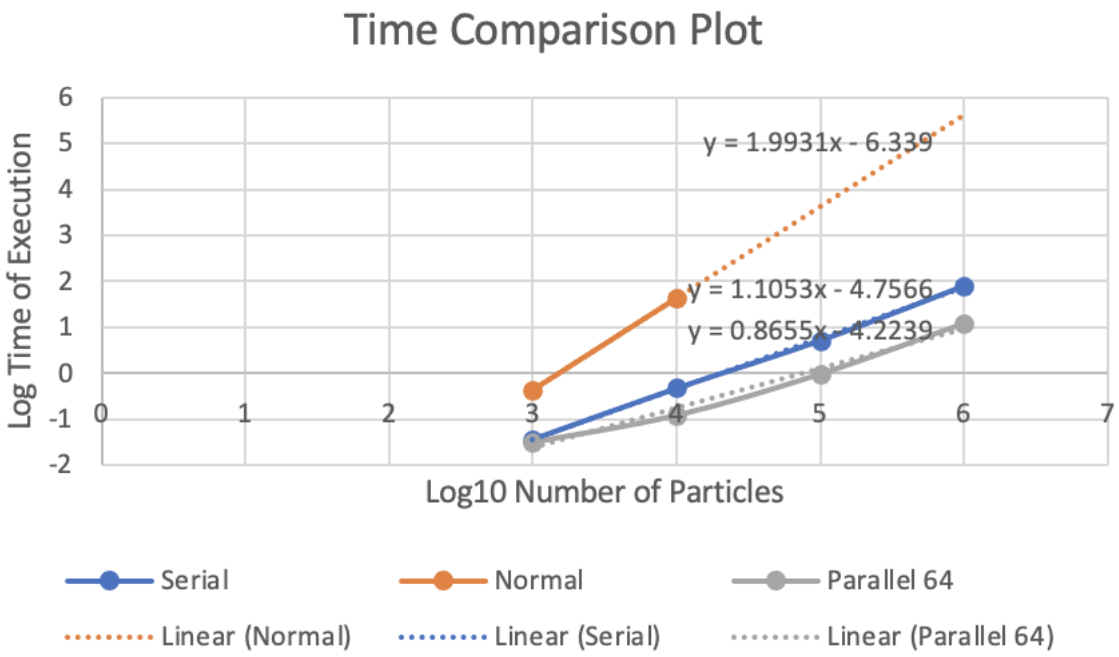

Figure 3: Final Peak Performance With Block Size 4

I tried many different data structures when it came to binning. One of the most interesting things I noticed when testing was that after about 250,000 particles, the slope of the line would change. I believe this is mainly due to the cache size of L3 being filled up and leading to worse memory bottlenecks. Since this will happen no matter the data structure, it was important to choose the right data structures for the job. Initially, I went with an unordered set because of the amortized constant lookup, insert, and delete of the particles. However, the space usage of the set was too large and led to severe cache misses and slow computation times. After watching a video of Bjarne talking about list vs. vector, it was interesting to learn that even in real-world applications, iterating through and deleting from a vector was fast because the underlying hardware was much more supportive of the underlying algorithm. As a result, even though a linked list could provide constant time lookup, it still leads to slower time compared to a vector because traversing through a linked list is much slower than a vector due to the memory hierarchy. As a result, a vector was the chosen data structure. With a vector of vectors, the memory is very compact, leading to better cache locality. Furthermore, given that I clear out the bins every time, inserting, lookup, and deleting are all very effective. Given that I used the built-in vector functions, many of the vector usages were all constant time operations.

Beyond vectors, I wanted to use dynamic arrays for the bins as well because the number of bins stays constant between each iteration. However, after testing both components, there were minimal improvements to the execution time. This could be due to the fact that the underlying array of vectors is an array. The main component of importance was the underlying hardware technology. Due to the fact that we need to store all the particles, keeping cache-friendly applications is very important to the execution of the algorithm. This is especially important when the cache begins to fill up, and not everything can fit into the cache. As a result, looping through the bins sequentially was very effective in keeping the elements within the cache. The one downside of keeping pointers to particles is that the particles within the bins may be spread randomly throughout the particle array, leading to memory not being in the cache. Unfortunately, this wasn't something I could change because the particles had to be kept within their original arrays.

5 Parallel Implementation

5.1 Shared Memory Synchronization Bottlenecks

Given that I'm doing equal but opposite forces rather than doing a single particle calculation along every bin, the acceleration component for force calculation did need some amount of synchronization control. The main issue is adjusting the acceleration of your neighbor particle. I fixed this by setting each update to the acceleration to an atomic update. One moment in time, I didn't need to use atomic updates because I executed the updates on every two rows of bins. This resulted in no possible race conditions for my code. However, the duplication of even and odd buckets led the compiler to build worse optimizations, which led me to the conclusion of putting both updates in a single loop, leading to a faster runtime.

Beyond the calculation step, the moving particles step took much longer for the particle simulation. This is because the moving step is harder to parallelize because there is a lot more synchronization required. When putting the particles get moved, we need to choose which bin these particles are in. Thus, different threads might be putting particles into the same bin, leading to race conditions. Therefore, the main synchronization technique I used was locking the bins when iterating over particles. Since the particles are uniformly distributed in the particle array, we lock and unlock the bins at some random rate. Thus, this leads to my times for moving particles being almost the same amount of time compared to the actual calculation component. I tried to minimize the amount of time I needed to synchronize my data. Without synchronization, it would have been orders of magnitude faster. However, due to the collisions between various bins, my performance decreased.

5.2 Parallel Specific Performance Optimizations

Prior to using atomics, I tried not to use atomics during the calculation step. This was possible because we only needed the bin to the right and the three bins below. As a result, if you have the parallel execution every second row, you will never have an update collision resulting in a data race. However, I tested both methods of atomics and spacing, but it seems like the atomic version ended up around 10 seconds faster. This could be due to the fact that the compiler was able to generate better code for the atomic version because the looping structure was more apparent to the compiler. Thus, this change allowed my parallel efficiency to improve by about 10%.

For the execution of the binning algorithm, I understood that each binning could be executed in parallel as long as the atomic updates to the particles persisted. Thus, I achieved high parallelization within the computation step of the algorithm. After the calculation step, I made sure to have a thread barrier before moving to the rebinning step. Within the rebinning step, the main optimizations were either looping through bins or looping through particles. I found that looping through particles was much quicker than bins. When looping through particles, the particles are randomly distributed between the multiple bins within the grid. As a result, even though I had to lock the bins for each particle, there was a good chance that there would be no contention between the threads. For looping through binning, there is a good chance that the particles being moved would need to be locked alongside nearby bins. Thus, it was better to do the heavy synchronization component with the least amount of contention possible. The major optimization of binning was by far the most important step to improve the parallelization of the algorithm.

6 Odd Behavior

The only odd behavior in the program is when there are too many threads running the program. In perlmutter, there are 64 cpus, but running the program on the command line enforces over 256 threads. This leads to extra evictions and contention between the threads which leads to this drastic decrease in performance after 64 threads.

7 Performance on Other Machines

On my personal machine, I have the same odd behavior if I enforce higher number of OMP threads. However, the runtime performance of my mac was also drastically lower than Perlmutter. This can be attributed to the reduced amount of cores on my computer compared to perlmutter. Thus, I saw a

performance decrease. Furthermore, the CPUs on my computer don't have the same specifications as perlmutter leading to high compute times.

8 Ideal Speedup and Can I do Better?

We should see nearly 100% weak and strong scaling for the ideal speedup. However, due to the speed of my serial application, the strong scaling effectiveness seems lower despite the 12 seconds and 20 seconds, respectively, for 64 and 16 threads. The reason behind this nonideal speedup is that my serial and parallel execution don't follow the iteration pattern. Due to the nature of memory collisions between threads, I had to make sure there wasn't any false sharing or data races between specific modules. However, this was unavoidable during the particle rebinning because particles might have shifted to nearby bins.

To do better, the main optimization for parallel might be to reduce false sharing within the initial set of particles as well as attempt methods to rebin. While the current way of rebinning clears all the particles, there may be more effective ways of rebinning that will only remove a few elements each cycle. These changes might result in fewer collisions between threads, leading to a much better, stronger scaling on higher particle numbers. Beyond the synchronization penalty, it might be beneficial to look into the cache miss ratio within the serial and parallel segments of the code; if there are many cache misses taking place, it will be beneficial to understand why we are getting worse performance. This is especially effective in areas around 250000 particles because the entire cache could be filled up with data that the program is no longer using. Thus, these changes could improve the overall performance of both the serial and parallel executions.

9 Speedup and Execution Plots

9.1 Strong Scaling

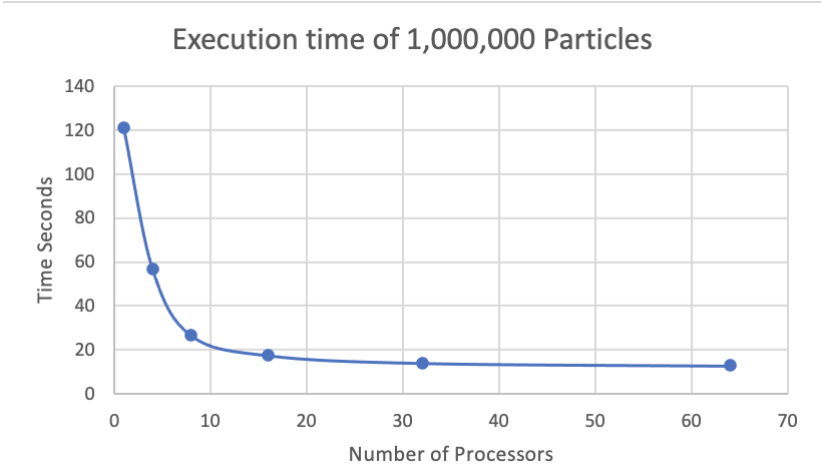

Figure 4: Execution time for Strong scaling at 1000000 particles

We see a sharp drop in execution time for the first few additional processors. However, as the number of processors increases, the percentage of improvement doesn't scale as much. This is due to the contention between multiple threads when they update the bins.

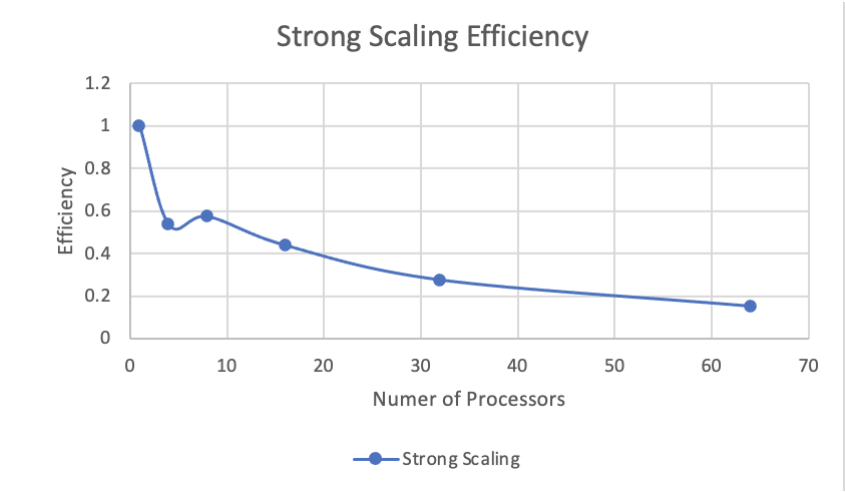

Figure 5: Execution time for Strong scaling at 1000000 particles

9.2 Weak Scaling

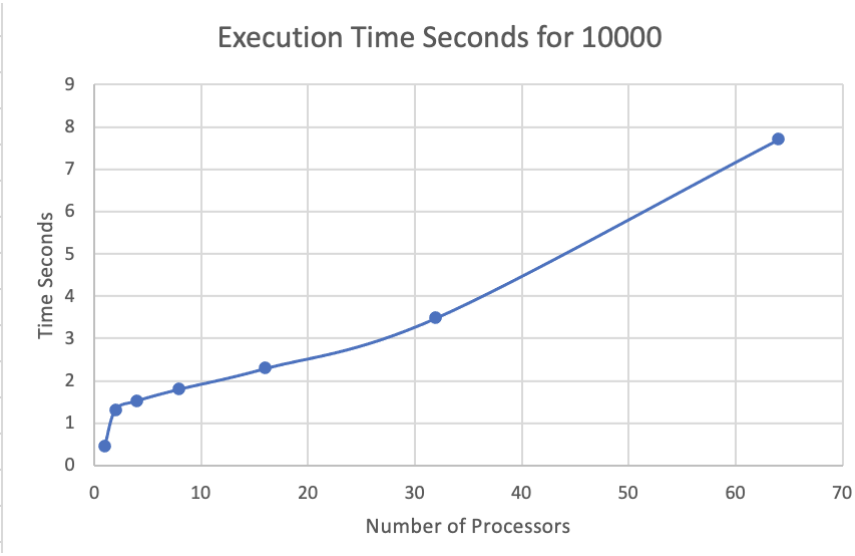

Figure 6: Execution time for Weak scaling at 10000 particles base

The time to complete does increase as the number of processors increases for the problem because of the synchronization overheads between bins. As the number of threads increases, the amount of contention increases.

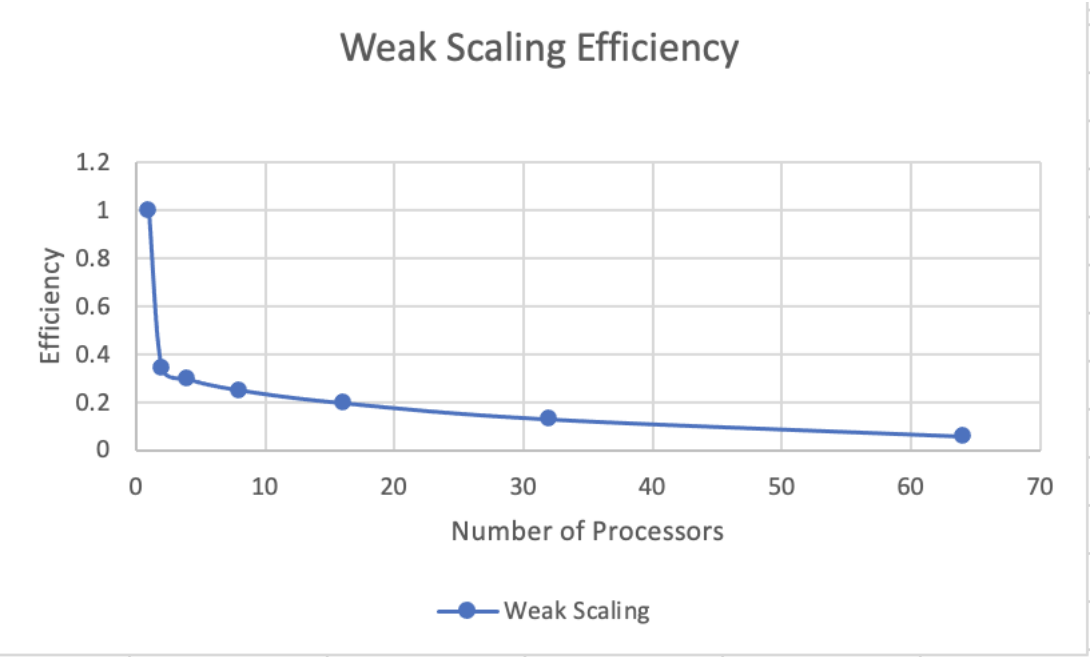

Figure 7: Weak Scaling Efficiency for 10000 particles base

10 Conclusion

Overall, as a first introduction to OpenMP, it was interesting to learn about how the fork-join model impacted the total execution time of the algorithm. Throughout the entire process, there were many times when segmentation faults would occur because I was accessing the wrong parts of memory. After going through multiple different executions of the parallel model, it was easier to be overly cautious with using locks and synchronization to make sure that the execution of the code remained correct. While I didn't see the highest possible parallel efficiency, it could be due to the fact that my serial execution was a lot better compared to the parallel execution. In a vacuum, my parallel execution does very well relative to a baseline of binning, which was around 200 seconds of execution. However, it can be hard to directly map the parallel efficiency because the methodology for serial vs. parallel isn't completely 1 to 1 because there are points where you need to synchronize your data that you otherwise wouldn't need to do within the serial implementation. Overall, the execution of the parallel implementation saw tremendous improvements in execution time and can be further improved upon depending on the tools available.