CS 267 Homework 2.1 Write-Up

Group:

11

Group members:

Josh Bedwell, Shirley Li, and Thomas Janas

Optimization Techniques:

Serialization:

The original implementation is at O(n^2), which we developed to an O(n) time complexity. We did this by adding a vector of bins in column major order to a Grid struct. Each bin signified an area of the grid and held a vector of each particle position as a pair of doubles. Our bin size implementation was chosen as twice the size of the cutoff, which ensured that for any one particle in a pair-wise calculation would be compared to only the particles in 4 of the nearby bins. The original particle force calculations are done with the particles in the original bin and the particles in the 3 adjacent bins based on where the particle is located in the original bin. We found this method to work very well with our serialization. One drawback would be that our bin sizes are determined by the cutoff, whereas it could be beneficial to allow for a custom size of the bins.

To minimize the logic detecting which bins are relevant to a particle, each bin is set to the size of twice the cutoff radius for each particle. With this sizing, only four bins will be relevant to each particle, and you can detect which ones just by seeing which quadrant each particle is within its bin. If you were having really densely packed bins you could reduce the size of the bins and then iterate through less particles by only considering the bins in a spherical shape around the bin the particle of interest is in. This would involve some more complicated relevant bin calculations, so we decided to stick to the four maximum intersections. The code for calculating the bin size (and thus grid size) is in serial.cpp #22 - #27.

There were a few potential strategies for filling the grid. The original strategy we used was filling the bins of the grid with copied `particle_t`. The performance was not bad. A second strategy was to fill it with `particle_t*` to do a trick to calculate half the forces. The `apply_force` function is called with two `particle_t`. The force applied on them is equal and opposite, but it is only applied to one particle. We could apply the force to both particles and avoid duplicating some calculations, but that would require only calling the function once for

each pair of particles and not the same pair in the other order. If we store the particles in the grid as pointers, we can only call `apply_force` when `particle_pointer1 < particle_pointer2` (compare their memory location). This strategy did lead to a small performance bump in the serial code, but was ultimately abandoned due to the potential complications in the openmp code.

In the end it was decided that the only point of the grid structure is to map positions of particles, thus instead of bins having `particle_t` or `particle_t*` they only have `std::pair<double, double>`. This also seemed to lead to a small performance improvement, but more importantly it simplified the role of the grid in our parallel implementation.

In order to avoid a bunch of moves and passing parameters (which probably would have been optimized out by the compiler, but whatever), a large section of the `simulate_one_step` code is calculating the relevant bins for each particle, which is rather simple. For each bin that might contain particles within the cutoff distance of the particle of interest, you can simply iterate through all the coordinates in it and apply the forces there.

OpenMP:

The key with the OpenMP implementation is to keep all threads as busy as possible while preventing any data races. The first and simplest optimization we made was adding `#pragma omp for` to the loop that computes forces on each particle (openmp.cpp #159), as well as the loop that updates their positions (openmp.cpp #233). With this done, the calculations of the simulation were effectively parallelized.

In order to build the Grid object without data races, that section was originally wrapped in a `#pragma omp single`, and all the work of sorting the particles into bins was done by one thread. Unfortunately this was not a negligible amount of work, and it caused the scaling to not work out at high numbers of particles as building the grid dominated the compute time. The next attempt to speed this up was to have each thread create a thread-local Grid and sort its fraction of the particles into grid bins, and then have each thread copy their bins into the global grid one at a time (using `#pragma omp critical`). Unfortunately this did not offload so much of the work to each thread since all the copying was still essentially single threaded. We needed to find a way to parallelize copying bins from threads to the global object.

The first attempt at parallel copying thread bins to global bins was to give each thread an index offset based on their thread number. At the first iteration, thread 0 would copy `thread.bins[0]` to `global.bins[0]`, thread 1 would copy `thread.bins[1]` to `global.bins[1]`, and so on. There was a `#pragma omp barrier` after each copy in the loop so that all threads could finish copying before moving on to the next index. The unfortunate part of this strategy was the amount of barriers & synchronizations required. For a 100x100 grid (10,000 bins) and 64

threads, there would be 640,000 synchronizations *per simulation step*. That was not the required performance benefit. The step to mitigate this was to instead of having each thread copy the contents of 1 bin before synchronizing and moving on, divide the number of bins by the number of threads to create as many regions as threads and have the threads each copy a region before moving on. In the case of the 100x100 grid with 64 threads, there are now only 10,048 synchronizations per simulation step to build the grid. This is still a very high number, but it's about as good as you can get while dividing the work evenly between the threads. At no point are any threads idle, synchronizations are minimized, and work is evenly divided. This work is observable in openmp.cpp #115- #147.

At this point every single line in `simulate_one_step` is run in parallel, with the only synchronizations happening at some `#pragma omp barrier` directives during the cooperative grid creation process and the beginning of the `#pragma omp for` directives. There are a few potential areas for improvement and opinionated design choices made.

The main potential area for improvement is in allocations at each iteration of `simulate_one_step`. Currently each thread instantiates a new Grid object at the beginning of each call to `simulate_one_step`. A grid is essentially a vector of vectors of coordinates (std::pair<double, double>), so for a 100x100 grid this would be an allocation of a 10,000 length vector of vectors (64,000 across all threads). In an attempt to fix this, we created a static vector of Grids with the same length as the number of threads in the `init_simulation` function, and then have each thread access the grid associated with their thread number. For some reason this caused a huge performance drop in our tests. It could be due to having to create another `#pragma omp parallel` region in the `init_simulation` function to get the number of threads, but that behavior still doesn't make very much sense.

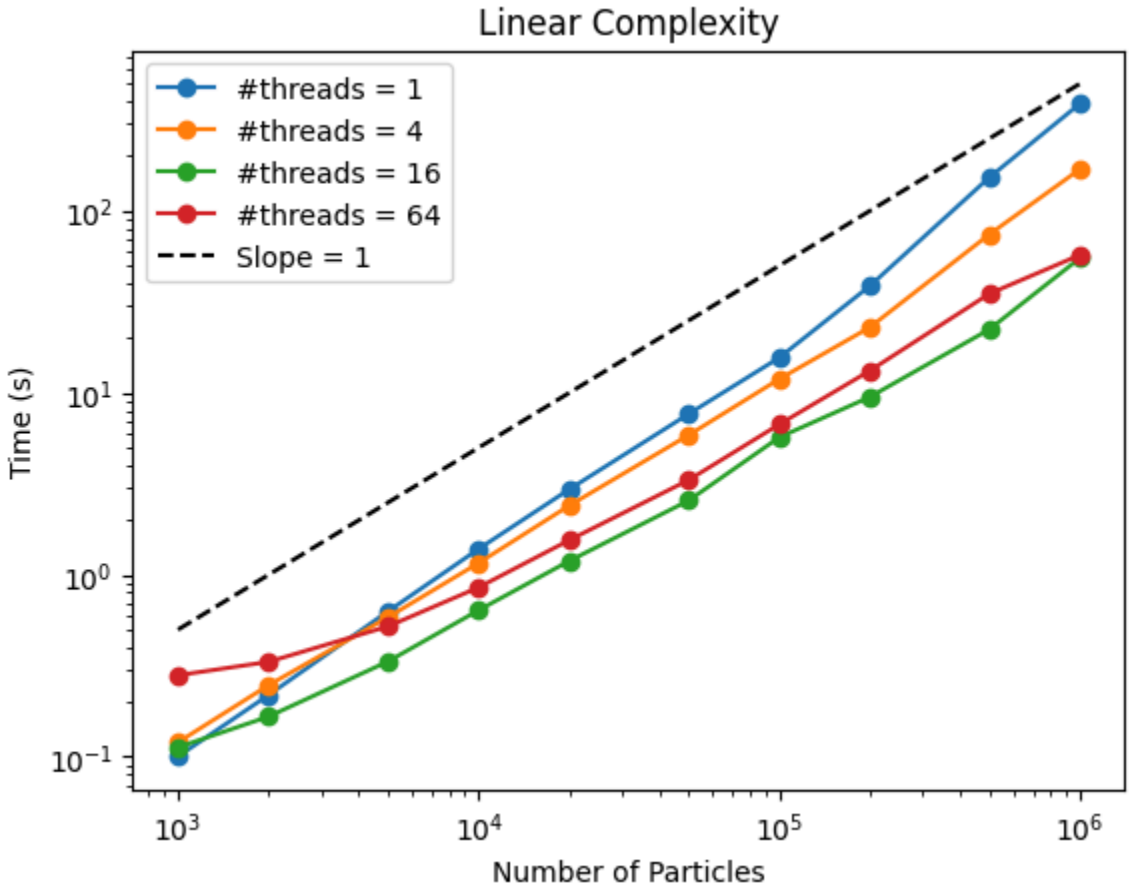
As discussed in the serial section, we opted to not attempt to apply the forces to both particles in `apply_force` due to the potential data races involved. You could have particle A interacting with particle B with both of their accelerations being updated, but simultaneously on another thread particle B interacting with particle C, and both of their accelerations being updated. Because of this complexity, we stuck to the strategy of the grid simply holding copies of particle positions so that applying the forces on the particles can be cleanly divided between the threads, with clear thread responsibility for each particle.

We talked to another group that tried to parallelize the x and y calculations. That sounded interesting, but our group avoided it because we already effectively utilize all threads and that would introduce a lot of complexity to the code.

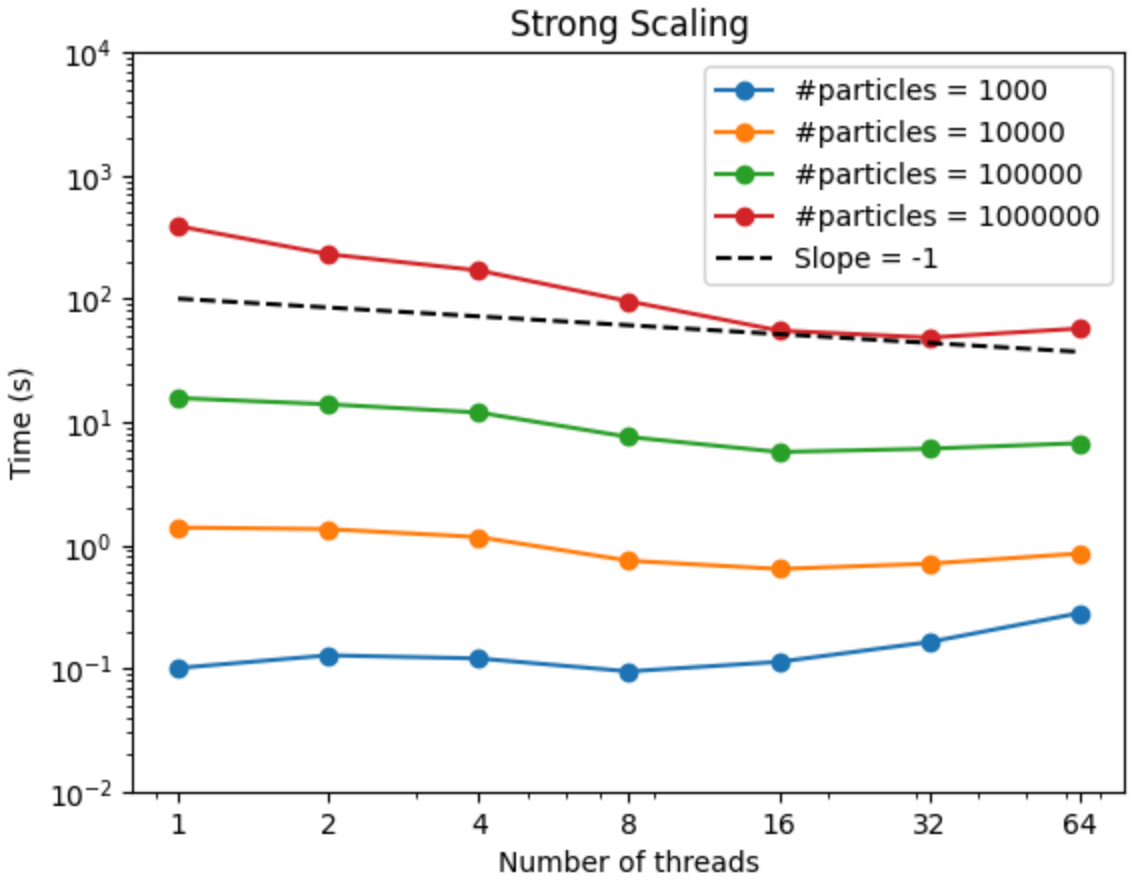
As is shown in the results, it appears that the Grid construction is the most hindering aspect of our OpenMP solution, preventing the program from fully utilizing high numbers of

threads. Some potential avenues to improve this would be capping the number of threads used when creating the grid. Some testing could find the optimal number of threads and that could be applied as a cap.

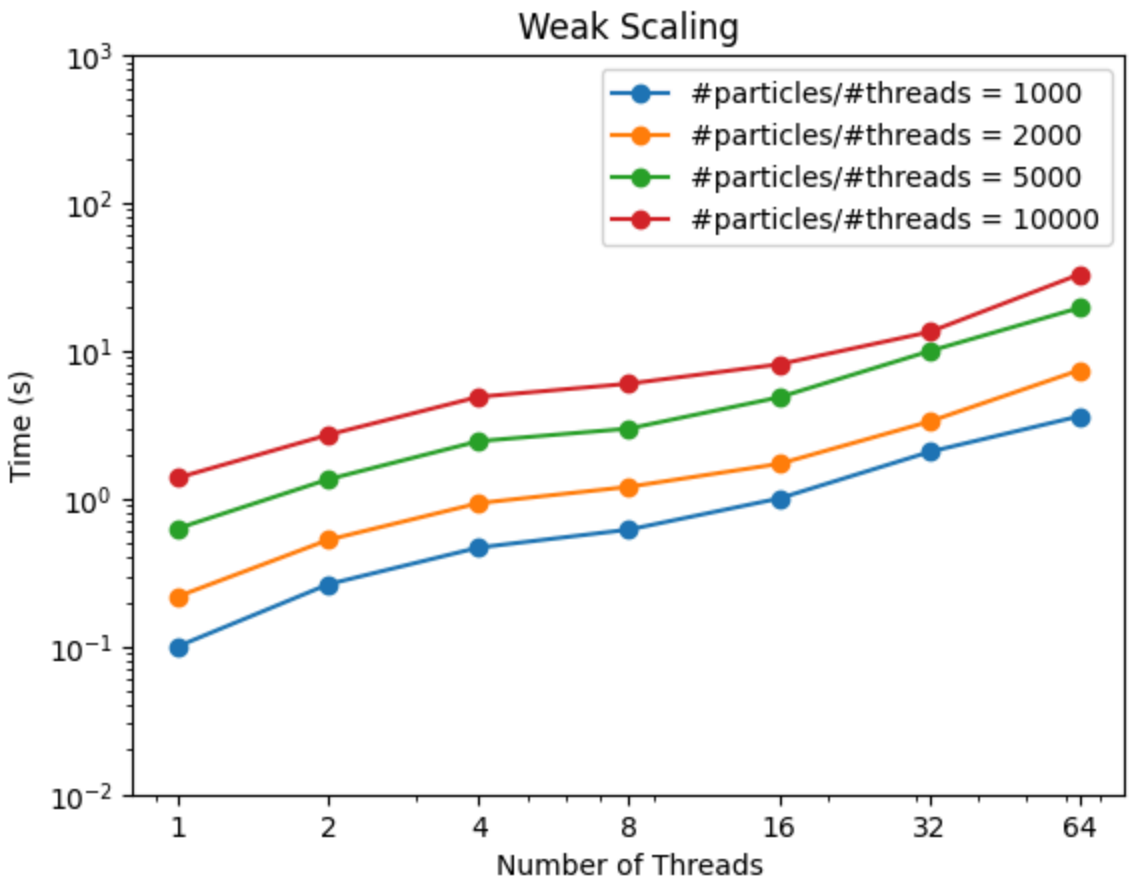
Results (recreate recitation plots with our results):



Our linear scaling plot shows that our solution successfully converts the base problem from O(n^2) to O(n). This plot is very similar to the example plot in the recitation slides, including the conflict between N=16 and 64 threads performance. The scaling of the problem at these higher thread counts looks a little less than linear, suggesting that the relative efficiency of these higher thread counts could increase at higher numbers, but thread counts higher than 16 are not fully utilized at this scale.



Our strong scaling, similar to the example plot, shows that our code benefits more from small increases in thread count. At these particle counts the performance benefits are fully realized at N=8 or 16. We also see a greater increase in performance at higher particle counts, likely due to the more effective utilization of the extra threads during the force calculations minimizing the cost of cooperatively constructing the grid.



Our graph (like the example plot) shows relatively poor weak scaling. This really highlights the fact that our extra threads are not being fully utilized. Again we see a sweet spot of improvement from N=4 to N=16, which again suggests that is the range where the synchronization costs of our algorithm are least significant compared to the work of calculating the actual particle simulation. The trend is roughly the same for each number of particles per thread, indicating that this issue is something that occurs no matter the number of particles (the grid construction).