CS267-HW2.1

Justin Kalloor, Yihua Zhou

February 2023

1 Problem Statement

The objective of this assignment is to first implement and then accelerate the particle simulation within a shared memory model. In the simulation, particles interact by repelling one another.

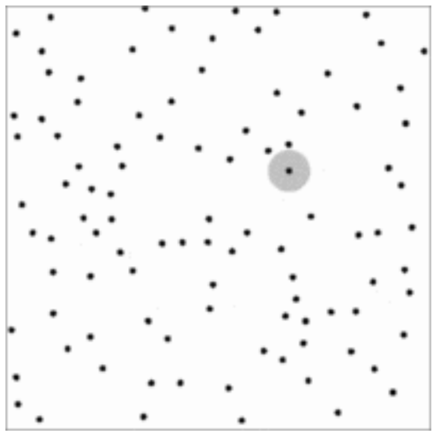

Figure 1: problem scheme

1.1 Starter Code

The naive code calculates the forces between all N particles at each step, causing a $O(N^2)$ time complexity.

1.2 Tasks

1. The first task is to decrease the time complexity of the simulation to $O(N)$.
2. The second task is to parallel the serial code using *openmp*, so that the code can be executed by multiple threads at one time.

2 Task 1 – serial code

The naive implementation calculates the force applied between every set of particles on each other. Therefore, you must calculate N^2 interactions total. Because the density of the particles is so small, we can actually have each particle look at its closest neighbors, and that number of neighbors will be around $O(1)$. Therefore, the total complexity can scale to $O(N)$ rather than $O(N^2)$.

To make the calculations $O(N)$ complexity, our implementation uses the following algorithm:

1. First, divide the area into bins.

2. Initialize the bins by assigning each particle to its corresponding bin.

3. Loop over all the bins.

   (a) For each bin, look at the adjacent 8 bins and itself (total up to 9 bins).

   (b) For each pair of bins, calculate the forces between each particle in the bins.

   (c) Since forces are symmetric, only need to calculate the force between every pair of particles once.

   Note here that the bin size to be chosen must be greater than $0.5 * cutoff$ in order to ensure correctness (every relevant neighbor for each particle in a bin is inside of the 8 neighboring bins).

4. After the force has been calculated for every particle, move each particle and update which bin each particle belongs to.

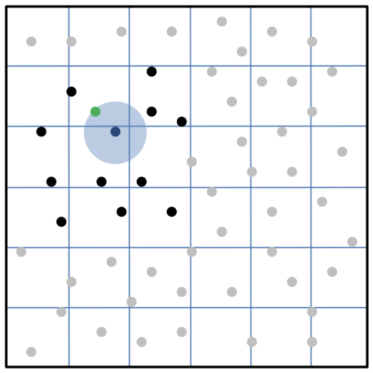

Figure 2: bin scheme

3 Data Structures

For the serial code, we used the following data structures:

1. The bins were implemented with a map of unordered sets. Each map entry mapped a bin id to the corresponding set of particle objects.

2. We used an unordered set to contain the particles in the bin because they are kept in a hash table which allows for $O(1)$ lookup, insertion, and removal. It is more costly to loop through the elements, but we found that having the fast look ups provided a much larger speedup over set or vector. In addition, the hashtable is continuos in memory, so if we can reserve a reasonable size space, we can work with

4 log-log scale plot

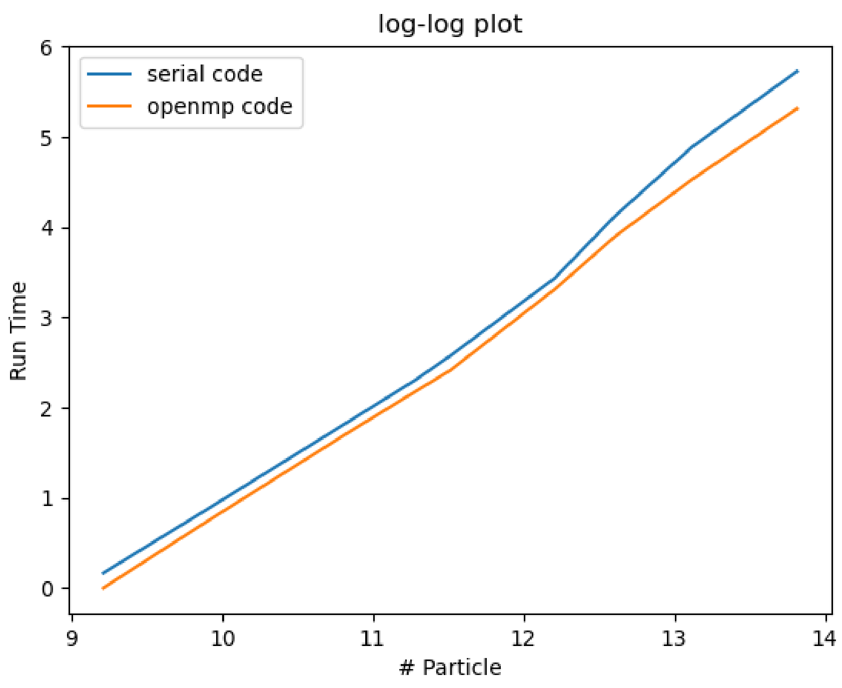

Figure 3: log-log plot

5 Parallelization and Synchronization methods

For our parallelization, we used OMP to quickly run our simulation. Since each particle can run independently, the problem becomes embarassingly parallel and we can even assign a separate processor for each bin. However, we are using shared memory, so there are some important data structures that must be synchronized in order to ensure correctness. Most notably, when particles move from one bin to another, this action must be synchronized.

5.1 pragma omp for

The most used parallelization compiler instruction we used was the for command in OMP. This tells the compiler to assign a different thread to each iteration of a for loop automatically. Therefore, in theory, the first iteration of the loop will be handled by thread 1, then the next iteration by thread 2, etc. This easily speeds up our processing and we see very good strong and weak scaling.

5.2 pragma omp atomic

When we calculate the acceleration of the particle, different threads may write to the same particle. So allowing one thread write to one particle one time by using pragma omp atomic can avoid data race.

5.3 pragma omp critical

pragma omp atomic has less overhead than pragma omp critical. Howerver, the pragma omp atomic can only be used on specific operations like add or subtract.
To move particle from original bins to new bins, can only use pragma omp critical to avoid data race.

5.4 locks

One of the drawbacks of the pragma omp critical is that it will lock the whole section of pragma omp critical region, which is like a big lock. And some parts of the code that does not need to be locked will be locked by pragma omp critical.
By using the locks, only related part will be locked and other part will continue working, which is more efficient than pragma omp critical.

6 Design Choices

We tried the following designs in order to speed up performance:

1. Realizing Force is symmetric: Our first speedup was algorithmic. We realized that the force applied from particle A to particle B is just the opposite of the force applied from particle B onto particle A. Therefore, we can avoid looping twice and only have to visit each relevant pair of particles once.

2. Varying Data structures: We tried several different data structures for the bins. We thought that for parallelization and cache efficiency, it would be good to have the data structure be contiguous memory so we tried both std::set and std::vector. This means that they can be easily brought into the cache and it is incredibl easy to iterate over all the elements in the set. However, both of these have the drawback that remove() is no longer O(1). This massive drawback heavily outweighed the positives, so in the end we decided to use std::unordered_set.

3. Varying Bin Size: Our first design choice was how to set the bin size as a function of the cutoff parameter. We knew for correctness that the bin size had to be at least 0.5· cutoff, but it is unclear after that what the ideal bin size should be. If the bin size is larger, then more particles can be interacted in one for loop, thus allowing for some cache efficiencies. However, if the bin size is too large, than there will be too many particles and lead to cache misses and slowdowns. We plot varying the bin size in 4 different lines, and determined that using 3· cutoff is the best size.

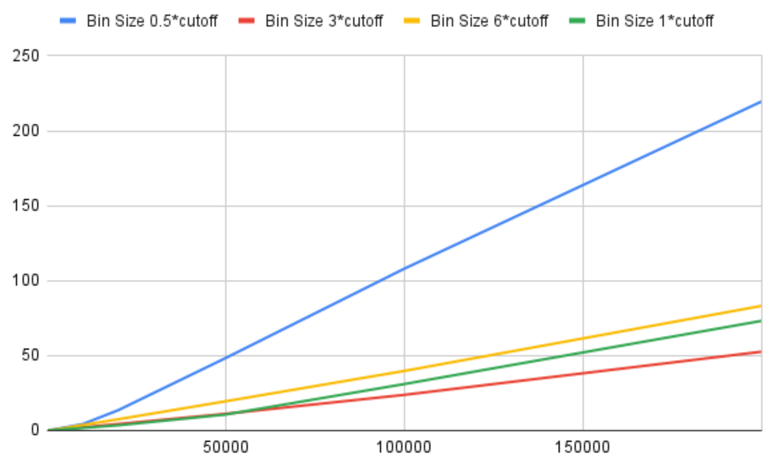

Figure 4: Performance scaling as we vary the bin size

4. Critical vs Locks: As dicussed earlier, we saw a major speedup when we switched from using omp critical to locks. This is because we no longer mark the entire bin data structure as critical, but rather only lock the bin entry that we need. Therefore, multiple threads can write to the bins at the same time, as long as they are separate bins.

7 Scaling

7.1 Strong Scaling

We tested the performance of our code in the scope of 1 million particles.

# threads	time(s)	efficiency
1	207	
2	205	
4	105.9	48.9%
8	58.9	44%
16	35.3	36.7%
32	27	24%
64	19.6	16.5%

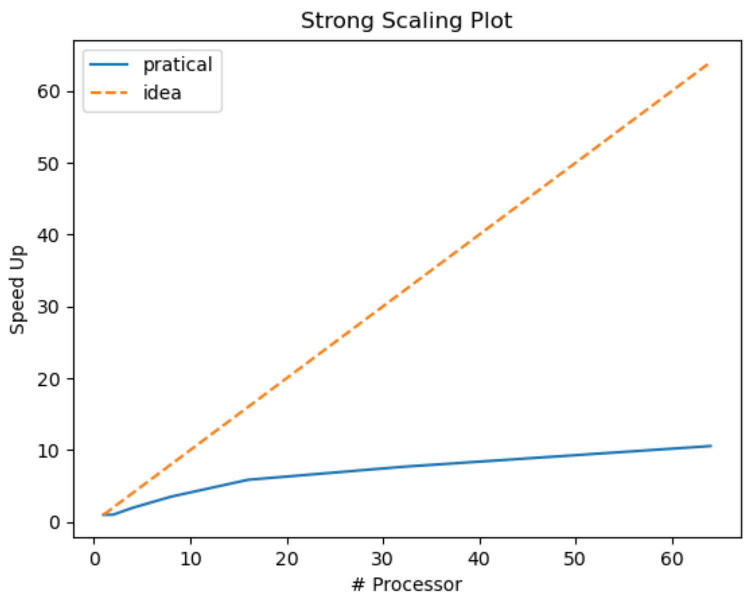

Figure 5: Strong scaling

7.2 Weak Scaling

# threads	# particle	time(s)
1	10k	1.04
2	20k	2.12
4	40k	2.26
8	80k	2.4
16	160k	2.76
32	320k	5.87
64	640k	9.83

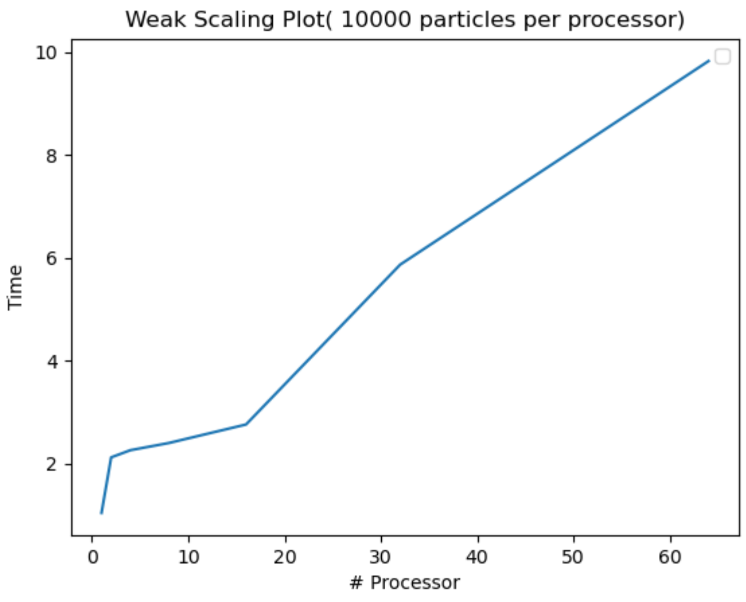

Figure 6: Strong scaling

8 discussion

8.1 no improvement from 1 thread to 2 threads

Interestingly, we do not see the improvement we expect when we go from 1 thread to 2 threads, but then scaling does continue normally after that. One idea we had for this was that the 2 threads competed for the same memory since they worked on adjacent bins. This means the memory conflicts would negate the benefit of the 2nd thread. However, when we tried blocking the loops into distinct sections rather than using pragma for, we saw no difference in the performance. So, we still need to understand why there is no difference in performance.

8.2 Difference between pragma omp critical and locks

At the initial implementation, we use pragma omp critical to tackle the issue of race conditions in the function of $move()$.
By using "pragma omp critical", the time of running 1M particles with 64 processors is about 50 seconds and increasing number of processor beyond 16 can not get any improvement.
By using locks, the time of running 1M particles with 64 processors is about 20s, which is definetely faster

than using ”pragma omp critical”.
So it is better to use locks rather than ”pragma omp critical”, which locks more regions than necessary.

9 Contributions

1. Justin - Modified the apply_force algorithm in order to see big speedup. Additionally, tried many different methods of speedup including bin_size, varying data structures, and blocking the array.
2. Yihua - Implemented the initial version of serial code. Using different ”omp primitives” to parallel the serial code that is optimized by Justin and achieved the performance of 20s (1M particles) with 64 processors.