

CS 267 HW 2-1

Parallelizing a Particle Simulation

Abstract: In this assignment, we implemented the particle simulation system, where each particle interacts by repelling one another. The naive solution is to compute the forces on the particles by iterating through every pair of particles, which runs in $O(N^2)$ asymptotic complexity. We first modify the serial running program in two different ways to achieve an $O(N)$ time complexity and secondly speed up to run close to time T/p using OpenMP.

Note: For this assignment, we present two different implementations based on [this help session](#): 1) deleting and rebinning every timestep, and 2) maintaining bins and moving particles.

Member & contribution

- **Bo Li:** Option 1
- **Jingran Zhou:** Option 2

From quadratic to linear

Because we expect $O(N)$ interactions among the particles, we will only compute the force coming from the neighborhood particles. Therefore, we split the whole region into smaller bins and check only particles in neighboring bins. Because the density is uniform, we can consider the number of particles in each bin to be a constant number. Therefore, the overall complexity becomes $O(9d \cdot N) = O(N)$.

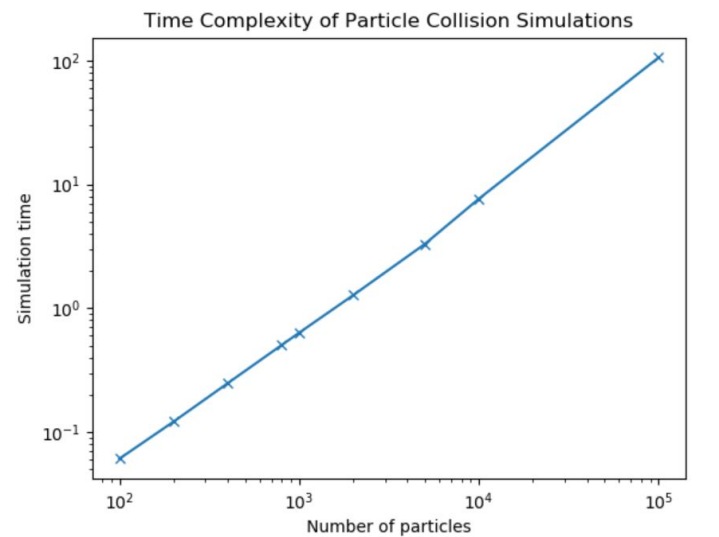
Option 1: Deleting bins and rebinning with vector & unordered_set

The implementation of Option 1 set each bin to be a container in the C++ standard library. In each simulation step, we will compute the index of the bin that each particle belongs to and put the particle in the correct bin position. Then we will compute the acceleration of particles by iterating each element in a single bin and apply the repelling force. When the acceleration and position of all particles have been updated, we will clear all the elements in the container at the end of the simulation step.

In this Option, we compare the performance of two different containers in the C++ standard library: `unordered_set` and `vector`. As shown from the below result, we notice that the `vector` performs much better than the `unordered_set`. By digging deeper into these two containers, we found that `unordered_set` is implemented by hashmap, which does not store data continuously in memory. By contrast, the memory is allocated continuously in `vector` and enables fast access when we iterate the elements. Therefore, we keep using the `vector` when we adjust the bin size.

Number of Particles	Naive	Unordered_Set	Vector
100	0.1188	0.0932	0.0610
200	0.4379	0.1937	0.1214
400	1.6771	0.3901	0.2479
800	6.5717	0.7908	0.5049
1000	10.2249	0.9845	0.6341
2000	40.5031	2.0071	1.2756
5000	251.802	5.4362	3.2971
10000	1006.37	15.8319	7.6529
100000	TLE	207.862	105.728

Below is the figure of simulation time versus the number of particles in the log-scale when we use vector as the data structure for a single bin.



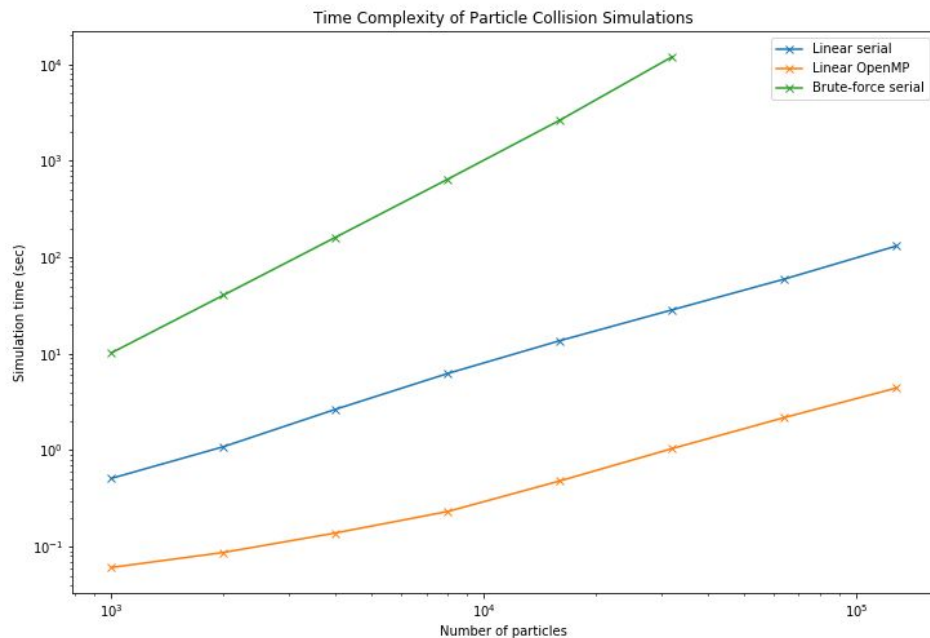
The above figure shows our algorithm is clearly linear.

Option 2: Maintaining bins & moving particles with unordered_set

The implementation of Option 2 has determined the optimal data structure to be `unordered_set`. Thanks to the assumption that the density of the particles is uniform, we can get away with quadtrees. Instead, we partition the particles into a matrix of equal-sized bins. During initialization, each particle is *inserted* into the corresponding bin based on its initial position. When applying forces, we *iterate over* particles in the bins to

simulate interactions. Finally, when we move the particles, if a particle should “jump” to a new bin, we *delete the particle* from the old bin and *insert* it into the new bin. Looking at the above operations, it is clear that we need a data structure that supports fast insertion, (unordered) iteration, and most importantly, *deletion by value* -- Let there be `unordered_set`.

If you are not convinced by the theoretical reasoning above, let experimental evidence speak for itself. Below is a log-log plot we obtained by running three implementations on various problem sizes. It shows our algorithm runs in $O(N)$ time (as opposed to $O(N^2)$).



The green line shows the brute-force serial implementation, which is naturally $O(N^2)$. It might not look quadratic in the above plot because the axes are on a log-scale. The blue line is the linear serial algorithm that we implemented, which is a drastic improvement compared with the quadratic algorithm. Lastly, the orange line is our OpenMP implementation with 68 threads, which is not only the fastest but also clearly linear.

Bin-level synchronization

To utilize the OpenMP to help us run the simulation in multi-threads. We use the command `#pragma omp` for in four different places, placing particles into bins (rebinning), applying force, updating particle positions and clearing bins. Because both `vector` and `unordered_set` provided by C++ STL, which is not thread-safe, we need to protect the matrix of bins against race conditions. However, *how* we offer this protection makes a significant difference in performance. Inspired by the lecture notes, we initially relied on **critical regions** (i.e., `#pragma omp critical`). While it worked, the performance clearly had room for improvement. After going to the Office Hour, we realized that critical regions are *very* expensive, which led us to **locks**. With a global lock,

we surround each write to the bins with `omp_set_lock()` and `omp_unset_lock()`. While it did offer some performance gains, this solution was not noticeably faster than using critical regions.

However, this parallel operation did not decrease the running time. As the figure shows, where we fix the number of particles to be 10,000, when the number of threads is over 10, the running time increases with the number of threads. This is a bad strong scaling.

```
>>> OpenMP [Strong Scaling] <<<
num thread = 1
Simulation Time = 8.67409 seconds for 10000 particles.
num thread = 5
Simulation Time = 5.10814 seconds for 10000 particles.
num thread = 10
Simulation Time = 10.2472 seconds for 10000 particles.
num thread = 17
Simulation Time = 16.2561 seconds for 10000 particles.
num thread = 34
Simulation Time = 22.8796 seconds for 10000 particles.
num thread = 68
Simulation Time = 35.6841 seconds for 10000 particles.
```

The mist in our heads faded a little after we devoured some udon. Since we are using *a single lock* to protect *all bin accesses*, whenever a thread writes to *any* bin, the whole matrix of bins are completely frozen. Therefore, it creates too many thread switches and race situations as we increase the number of threads, which is inefficient. Ideally, when a thread is accessing one bin, other threads should be able to access any other bins. Therefore, we should *protect each bin with its own lock*, which we call “bin-level locking.” Although we had to incur the overhead of initializing the locks, it proved to be more than worthwhile: **bin-level locking reduced the runtime almost 10 times.**

```
>>> OpenMP [Strong Scaling] <<<
num thread = 1
Simulation Time = 10.5582 seconds for 10000 particles.
num thread = 5
Simulation Time = 3.96842 seconds for 10000 particles.
num thread = 10
Simulation Time = 2.85748 seconds for 10000 particles.
num thread = 17
Simulation Time = 1.82911 seconds for 10000 particles.
num thread = 34
Simulation Time = 1.23061 seconds for 10000 particles.
num thread = 68
Simulation Time = 0.705497 seconds for 10000 particles.
```

Design Decisions

Bin size vs. bin count

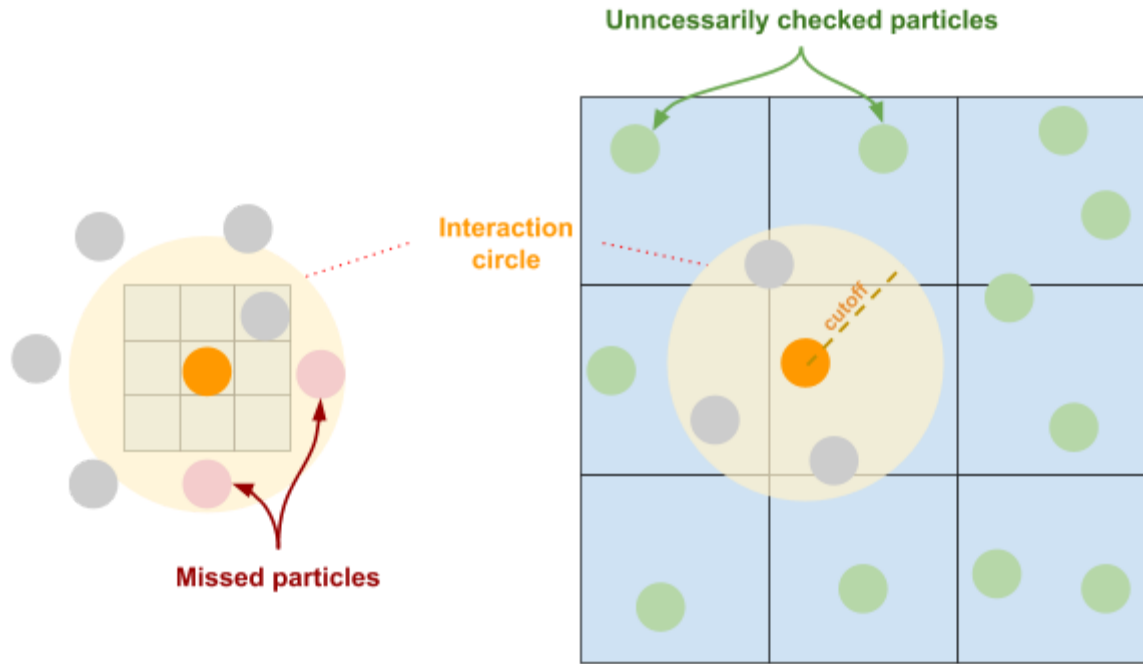
The bin size and the number of bins are two sides of the same coin, or so it seems. In the beginning, we chose to tune the bin count. However, after careful analysis, we realized we have to choose **bin size** as the tuning parameter. It turns out whether we choose to tune the bin size or the bin count *does* matter, as shown below.

Suppose we choose bin count as the tuning parameter. In other words, a line that looks like `#define BIN_CNT 0.01` appears near the top of our code. Naturally, we then calculate the bin size dynamically as `BinSize = double(size / BIN_CNT)`. Therefore, we have $BinSize \propto size$, where *size* is the length of the side of the square that bounds all the points. Further, an examination of `main.cpp` and `common.h` reveals an important truth: *the density of the particles is constant*. If the number of particles grows, the size of the square automatically expands to accommodate. This implies $N \propto size^2$. Let the density constant be ρ . Since the distribution of particles is uniform, we can calculate the average number of particles per grid $d = \rho \cdot BinSize^2 \propto size^2 \propto N$. Notice the time complexity of our algorithm degrades into $O(9d \cdot N) = O(N^2)$, which is no longer linear. By contrast, if we fix bin size as the tuning parameter, the average number of particles in a grid $d = \rho \cdot BinSize^2$ is a constant. Consequently, the time complexity $O(9d \cdot N) = O(N)$ is still desirably linear.

In conclusion, if we want to maintain a linear time complexity, we shall pick **bin size** instead of bin count as the fixed tuning parameter.

Optimal bin size

The neighboring bins of a particle define the region in which we compute interactions. And choosing the bin size, like most decisions in life, is a *tradeoff*. As shown on the left in the figure below, if the bin size is too small, the blue neighboring bins of the orange particle are effectively encased by the yellow interaction circle, thus we might accidentally ignore the effects of out-of-the-region particles (in red); However, if the bin size is too large, as depicted on the right, we might unnecessarily check many green particles that are too far away from the interaction circle, resulting in computational overhead.



Bin Size Too Small

Bin Size Too Large

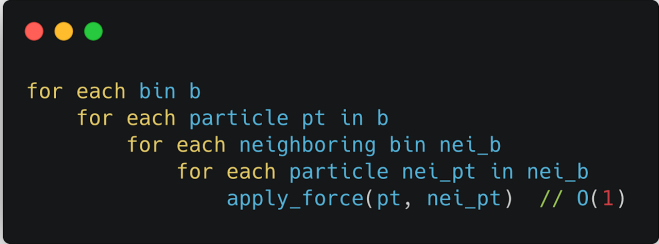
Therefore, the optimal bin size b , theoretically, should satisfy the equation $3b = 2 \cdot cutoff$. Solving the equation yields $\hat{b} = 0.0067$. To mitigate floating-point errors, we eventually chose a slightly larger bin size at 0.01, which has been proven by the below experiments to work reasonably well.

Particle Count \ Bin Size	0.01	0.02	0.05
100	0.0565	0.0610	0.0852
200	0.1154	0.1214	0.1811
400	0.2338	0.2479	0.3652
800	0.4703	0.5049	0.7583
1000	0.5869	0.6341	0.9623
2000	1.1870	1.2756	1.9415
5000	3.4637	3.2971	4.9710
10000	9.1207	7.6529	10.0916
100000	107.476	105.728	121.231

Degree of parallelism

The particle simulation mainly consists of two parts: `apply_force()` and `move()`. The latter is relatively simple because it has merely one loop, while the former has *four* nested loops. When using OpenMP to divide and distribute work among threads, the important idea is to make sure each thread has enough work to do, so as to dwarf the overhead of launching a thread. Thus, for each level of the nested loops, we need to carefully determine if extra parallelism is beneficial.

The pseudocode below shows the four nested loops for calculating interaction forces. For each of the four levels, we can parallelize (e.g., adding an extra `#pragma omp parallel for`) it or not. The outermost loop is parallelized by default, which is reasonable because each thread has enough work -- to process three levels of looping. By contrast, it would be a bad idea to parallelize the innermost loop, because `apply_force()` is a simple constant-time function, which is not enough work to dominate thread-launching costs. This leaves the middle two levels in question: should we parallelize them or not?



```
for each bin b
  for each particle pt in b
    for each neighboring bin nei_b
      for each particle nei_pt in nei_b
        apply_force(pt, nei_pt) // O(1)
```

Since there were only four possible scenarios, we employed an exhaustive search. After trying turning on/off parallelism on these two levels, we discovered that the best approach is the simplest: parallelizing only the outermost loop turned out to be the most effective.

Besides, we also compare the performance when each thread processes an entire row of bins versus the case that each thread processes only one bin, by using `#pragma omp for collapse(2)`. We can see that there is a little improvement if each thread processes one bin when the number of threads increases.

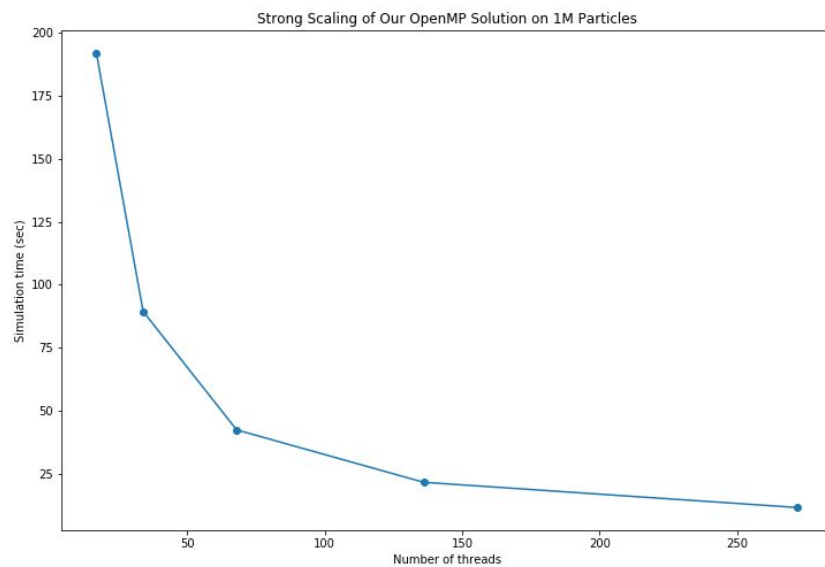
Number of Threads	<code>#pragma omp for</code>	<code>#pragma omp for collapse(2)</code>
1	10.5582 s	10.7674 s
5	3.9684 s	3.9624 s
10	2.8575 s	2.6069 s
17	1.8291 s	1.6803 s

34	1.2306 s	0.9453 s
68	0.7055 s	0.5483 s

Faster and less furious

Strong scaling

Our OpenMP implementation exhibits **strong scaling**. With a fixed problem size of 1,000,000 particles, we increase the number of threads, which produced the below figure.



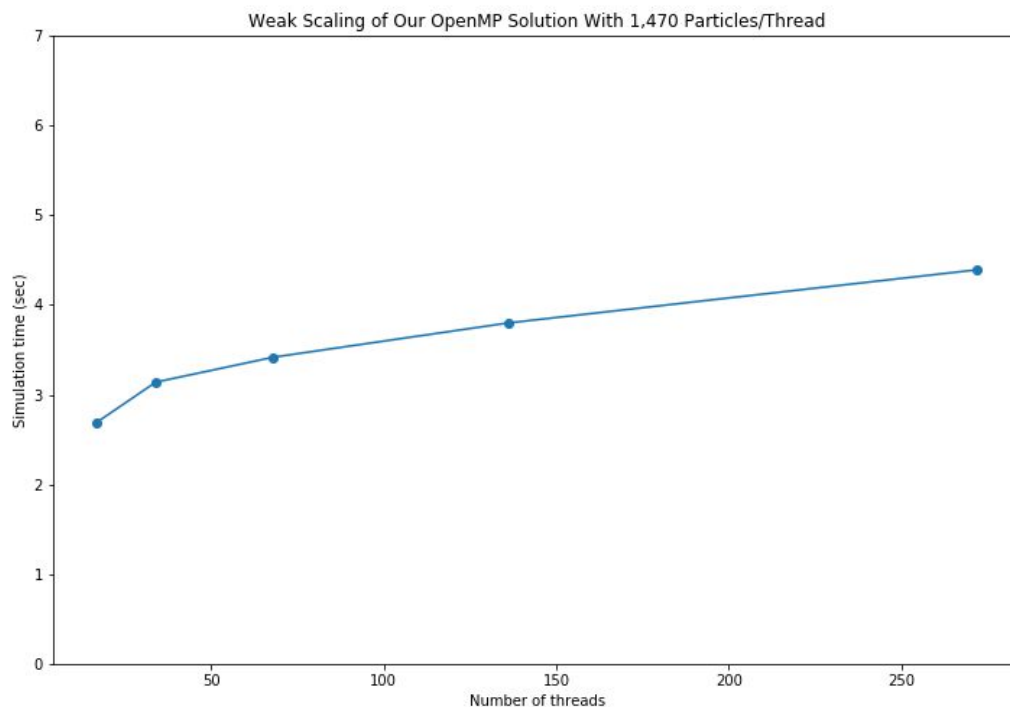
The above figure makes sense because we expect the runtime to be $O(N/p)$, where N is the fixed problem size and p is the number of threads. More quantitatively, the table below demonstrates the effect of strong scaling.

Number of threads	Runtime (sec)
17	191.804
34	89.4106
68	42.4856
136	21.7177
272	11.7018

Note that each time we double the number of threads, the runtime halves. For example, when we increase the number of threads from 136 to 272, the runtime decreases from around 21.7 s to 11.7 s.

Weak scaling

In addition, our solution also demonstrates **weak scaling**. With a fixed particle-to-thread ratio, we increase the number of threads and particles proportionally. Ideally, we should obtain a horizontal line, but in reality, we have the below figure.



We observe that as we increase the number of threads, even if the number of particles to be handled per thread is the same, the simulation time is not strictly constant. On the contrary, it is slightly increasing. One likely explanation of this phenomenon is the overhead of thread creation. Creating more threads is obviously more expensive than fewer ones.

Future work

While working on `apply_force()`, we discovered much redundant computation. Specifically, *when calculating the interaction between two particles, we only apply the force on one*, which is a potential source for optimization. Nonetheless, we cannot naively apply the force both ways, because it would eventually result in duplicate applications of the force. This observation could potentially open a door to a novel “Option 3” that is algorithmically different from the two we tried. Unfortunately, given the time limit, we had to forgo this opportunity.

Where does the time go?

We split the runtime into four different parts (i.e., rebinning, computing apply force, updating particle positions and clearing the bins). In both the serial and OpenMP version for option1, we measured the runtime of each part when running 100,000 particles.

	Rebinning	Apply Force	Update Position	Clear Bins	Total
Serial	10.7661	57.3011	2.0114	0.0652	70.149
1 Thread	26.4058	105.572	3.3632	0.1054	135.45
17 Threads	3.3076	13.9058	0.4973	0.0263	17.7559
68 Threads	1.0696	3.4576	0.1300	0.0159	4.6997
136 Threads	0.5969	1.7147	0.0708	0.0150	2.4291

We can see that most of the runtime goes to the “Apply Force” part. It is reasonable because in this step we need to compute the force from the neighboring particles and involve many computations and data reading and writing operations. Besides, as the number of threads increases, all parts runtime decreases proportionally. The most significant decrease comes from “Apply Force.”