CSC367 Project Phase 1 Report

Optimized Serial Implementation:

The whole implementation can be summarized into two parts: **Initialization** and **Simulation**.

For the **initialization** part, besides the starter code for initializing the particles and setting the size of the map. We also segregated the whole map into many small "grids" in 2D for optimization purpose with the following line of code:

```
Grid grids[length * length];
```

The class **Grid** is a linked list representing a small area of the whole map and its following nodes represent the particles located in this area. We collect all these grids in an array.

Suppose n is the number of particles in the simulation.

In the **simulation** part, we optimize the code for computing the collision forces which originally had a runtime of O(n^2). Our implementation executes in the following 3 steps:

1. Allocate every particle to its belonging grid.
2. For each grid, check which particles are contained in it. Then for each of its particles, compute the collision force.
3. Free the allocated memory in each grid for the next iteration of the simulation.

For the first step, each grid has its row and column index to represent its location on the map. Given the location of a particle on the *x* and *y* axis, we can calculate the grid index that this particle locates at. Since we need to go through each particle and put the particle into a grid, this step runs in O(n).

For the second step, each particle only needs to compute the collision force with the other particles in the nearby 3x3 grids. We no longer need to compute the force of all pairs of particles. Hence the runtime for this step is linear.

Since each step above runs in linear time, our optimized version of the serial implementation runs in O(n). We will show the actual running time is linear in the experiments and results section below.

Optimized OpenMP Implementation:

For the class **Grid**, we first ensure the thread safety of our Linked List data structure . The intuition is to make the "add_node" operation atomic. Using this approach, when one thread is writing to a grid, the others can't interrupt.

Hence, we make each Linked List Node wrapped with C++ "atomic" and enhance the Linked List to **Lock-Free Linked List**. Lock-Free programming is a technique in C++ programming. The idea is that when an operation is really small, we can make it atomic (spinlock) instead of using the mutex lock.

Other than the part we discussed above, most of the implementations are the same as our serial implementation. The process of **simulation** computation is the same except that we parallelized each of the 3 steps.

1. Statically partition the particles, and allocate every particle to its belonging grid in parallel.
2. Statically partition the grids by row major. Each thread does the particle collision computation in the grids.
3. Statically partition the grids by row major. Each thread frees the heap memory in grids.

Design Decisions:

We came up with multiple technical solutions at the beginning. We considered various aspects, especially the performance, before we decided on the final implementation.

Serial Optimization

We realize that the key to have a linear runtime is to eliminate unnecessary collision checks. In order words, we need to make each particle to only compare with the particles close to it, so it won't have to check all the particles in the whole map anymore.

Grid width

The number of grids is dependent on the size of the whole map and the **grid width**. For the same map size, we can adjust the **grid width** as a parameter to control the number of grids. Since our implementation lets each particle only compute collision force within the nearby 3x3 grids, the minimum value that **grid width** can be set is the **cutoff** which is the collision radius of a particle. We can increase the **grid width** by multiplying a scalar thus the whole map will have less number of grids. However, after testing with multiple scalar values, we conclude that increasing the **grid width** only causes a negative effect to the performance of our implementation.

Grid Data Structure

We decide to let each **Grid** store the particle information by linked list instead of a list. We know that the linked list has a fast write operation performance, but poor read operation performance because the nodes aren't consecutively stored in the memory which causes spatial locality issues. The list has a fast read operation performance but poor write operation performance.

We choose the linked list data structure because we distribute the particles almost uniformly in the map. The number of particles contained in each grid is typically small. As a result, the poor read operation performance won't affect that much. Also by using the linked list data structure, we can use atomic operation instead of lock to ensure thread safety while maintaining a decent performance.

Grids Flat Array

We store all the grids in a 1D array instead of a 2D nested array, then retrieve the grid by the formula:

grid_index = row_index * length + column index.

Collecting the grids in 1D array is more cache friendly which ultimately leads to better performance.

Particle Collision Computation Order

We had two options for this decision:

1. For loop through the particles array, for each particle, find it's locating grid, then compute the collision force by finding other particles in nearby 3x3 grids.
2. For loop through the grids array. For each grid, find all the particles in this grid, and compute the collision force for each particle in the grid.

The decision here mostly affects the parallel performance. We experimented with both options, and observed that the second option has better performance especially in the parallel context.

Concurrent Linked List

When two threads want to add nodes into the same grid, race conditions occur because a linked list isn't thread safe. We had two options to solve this issue:

1. For each grid, we initialize a mutex lock to make sure two threads won't write to the same grid simultaneously.
2. For each grid, make the add_node operation to be atomic. So when one thread is writing to a grid, the others can't interrupt.

We observed that the atomic add_node operation is a lot faster than using mutex lock. In fact, in our parallel implementation, we improved the Linked List to Lock-Free Linked List. As we have already discussed in the implementation section, Lock-Free programming is a technique in C++ programming. The idea is that when an operation is really small, we can make it atomic (spinlock) instead of using the mutex lock. Using this lock-free approach is advantageous since frequently calling “lock” and “unlock” is expensive and time-wasting.

Experiments and Results for: Serial

Below are the results we got by running “auto-teach-serial-small” and “auto-teach-serial-large”:

Serial-Small:

Particle Size	Running Time (S)
500	0.086799
1000	0.173988
1500	0.263326
2000	0.351349
3000	0.537733
4000	0.72304
6000	1.10093
8000	1.47996

Serial code is O(N^slope)
Slope estimates are : 1.003237 1.022052 1.002451 1.049615 1.029270 1.036947 1.028426

Slope estimate for line fit is: **1.025074**

Serial-Large:

Particle Size	Running Time (S)
10000	1.86432
20000	3.77493
40000	7.75038
80000	15.5024
160000	32.1529

Serial code is O(N^slope)

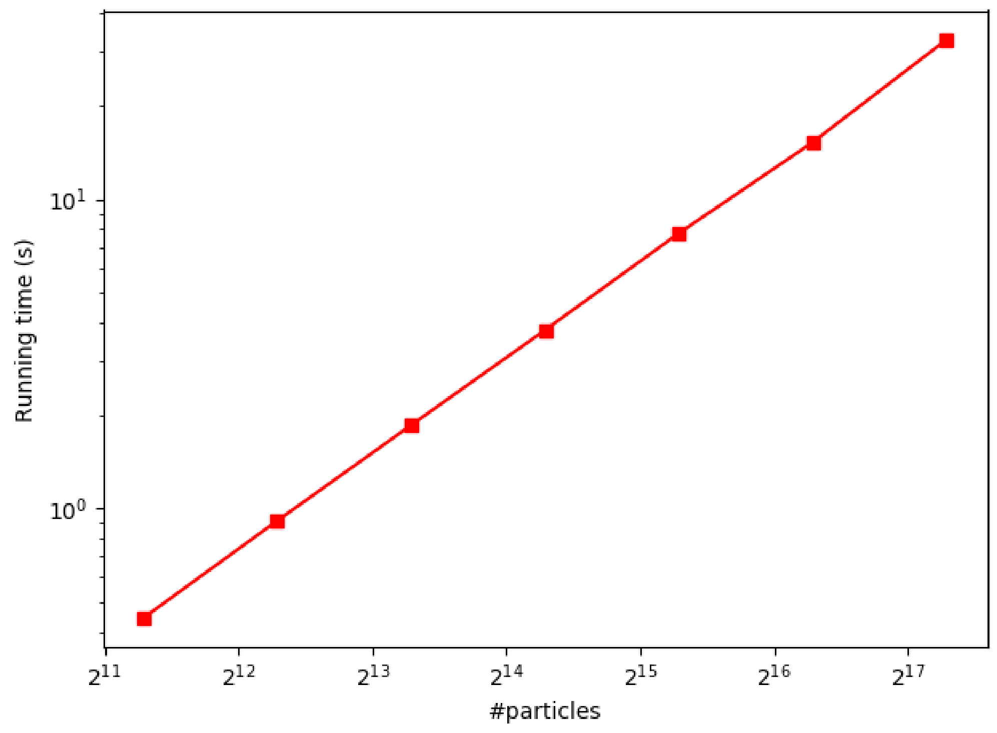
Slope estimates are : 1.017800 1.037817 1.000153 1.052457

Slope estimate for line fit is: **1.025442**

To draw an illustrative log-log plot of running time vs the number of particles with both small particles' size and large particles' size, we did an extra experiment and tested our serial implementation with the following number of particles.

Particle Size	Running Time (S)
2500	0.442857
5000	0.913073
10000	1.86051
20000	3.77486
40000	7.76081
80000	15.2815
160000	32.8195

Serial Log-log plot



Analysis:

Suppose n is the number of particles in the simulation.

According to our results we got by running "auto-teach-serial-small" and "auto-teach-serial-large", the slope estimate is about 1.025 ≈ 1, which means that the actual running time of our implementation is very close to O(n)

Based on the log-log plot above of running time vs the number of particles, the data points in the graph clearly form a straight line. This straight line indicates that our optimized version of the serial implementation runs in linear time (O(n)).

Experiments and Results for: OpenMP

For the **strong scaling**, we keep the particle size constant and vary the number of threads to be 1, 2, 4, 8 and 16. For the small dataset, we set this particle size to be 2000. For the large dataset, we set this particle size to be 10000.

For the **weak scaling**, we increase the problem size proportionally with the number of threads. For the small dataset, we set the particle size to be 2000, 4000, 8000, 16000 and 32000, while the corresponding number of threads are 1, 2, 4, 8 and 16. For the large dataset, we set the particle size to be 10000, 20000, 40000, 80000 and 160000, while the corresponding number of threads are 1, 2, 4, 8 and 16.

Below are the results we got:

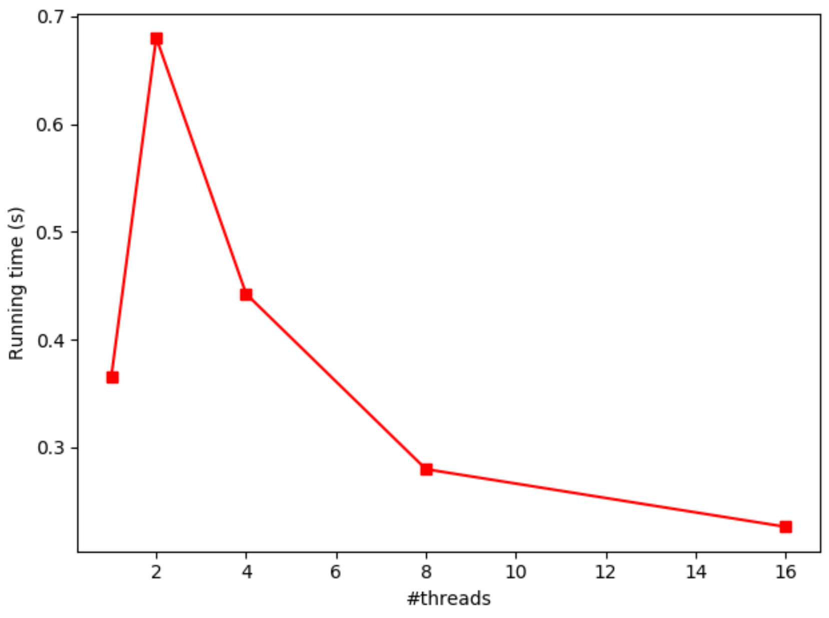
Our OMP implementation can finish the simulation for 160000 particles **around 11 seconds** with 16 threads, which is about the same as the reference performance provided in the handout.

- A table that shows the average weak and strong scaling efficiency of our optimized OpenMP implementation.

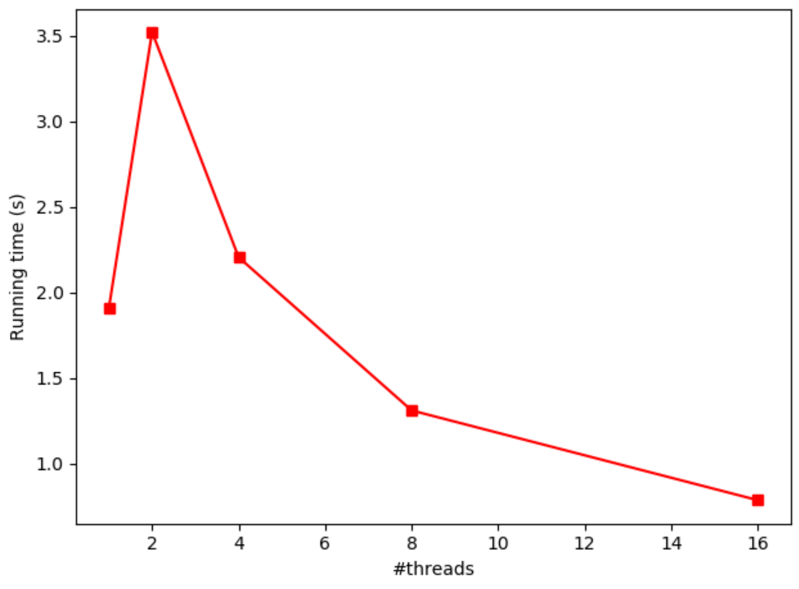
	Average strong scaling efficiency	Average weak scaling efficiency
Small Dataset	0.35	0.36
Large Dataset	0.37	0.37

- The strong and weak scaling plots that show the running time of our optimized OpenMP implementation.

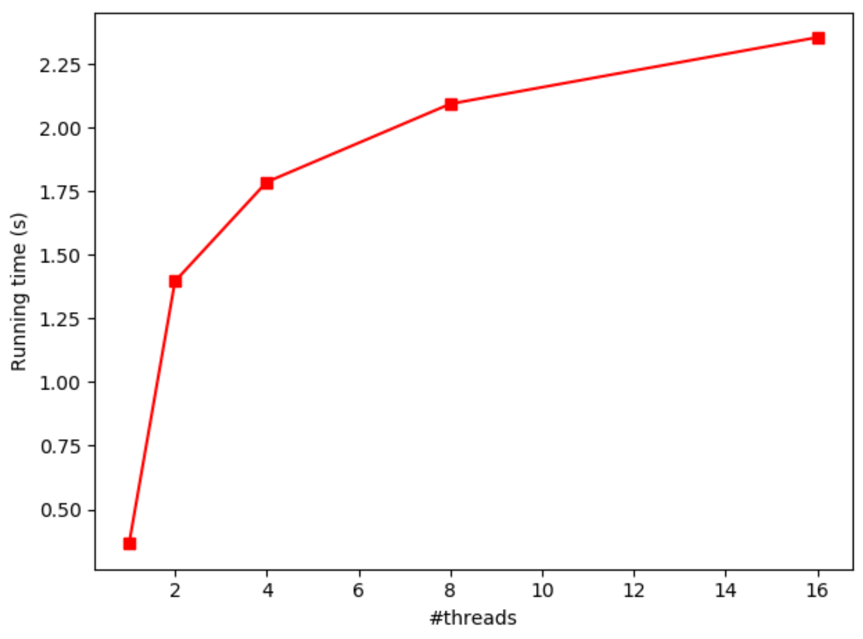
Strong scaling plot of running time for the small dataset



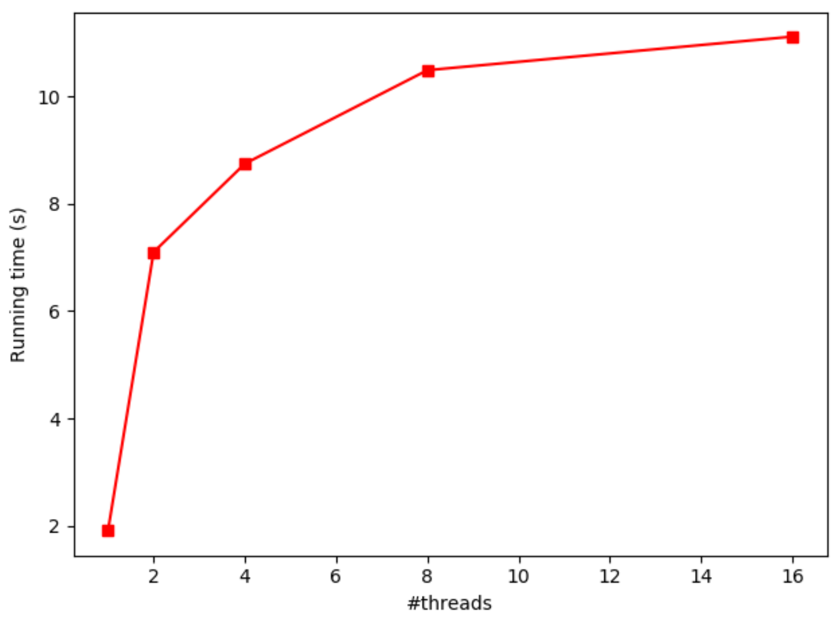
Strong scaling plot of running time for the large dataset



Weak scaling plot of running time for the small dataset

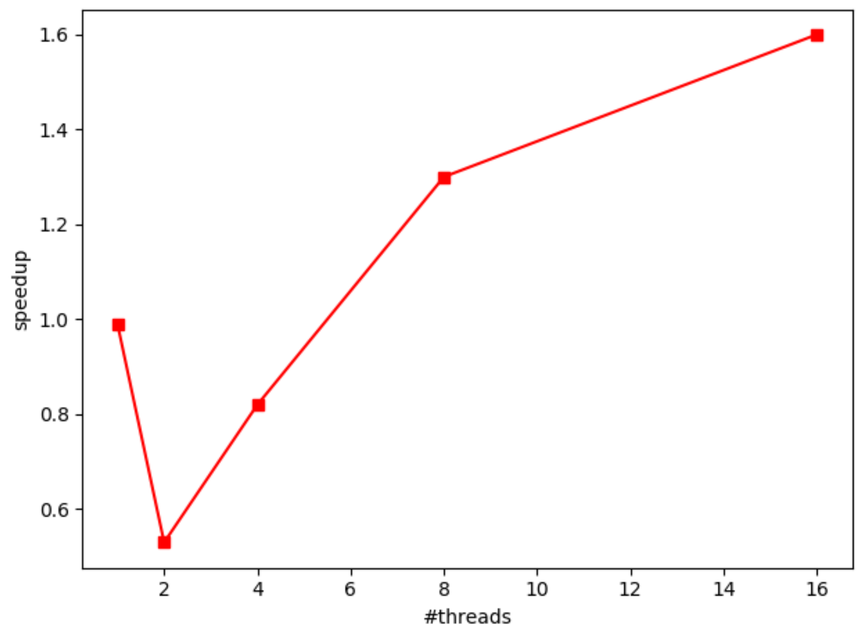


Weak scaling plot of running time for the large dataset

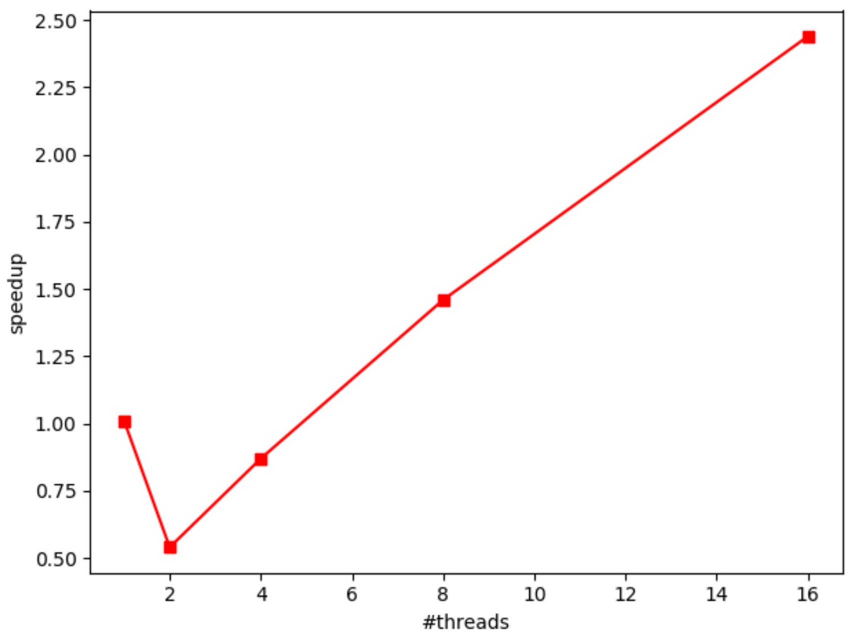


- The strong scaling plots that show the speedup of our optimized OpenMP implementation.

Strong scaling plot of speedup for the small dataset

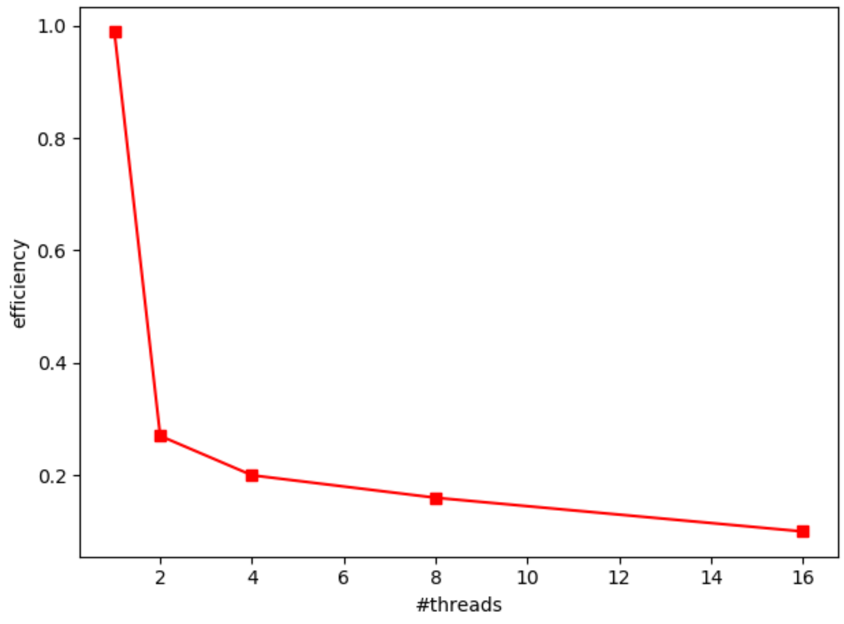


Strong scaling plot of speedup for the large dataset

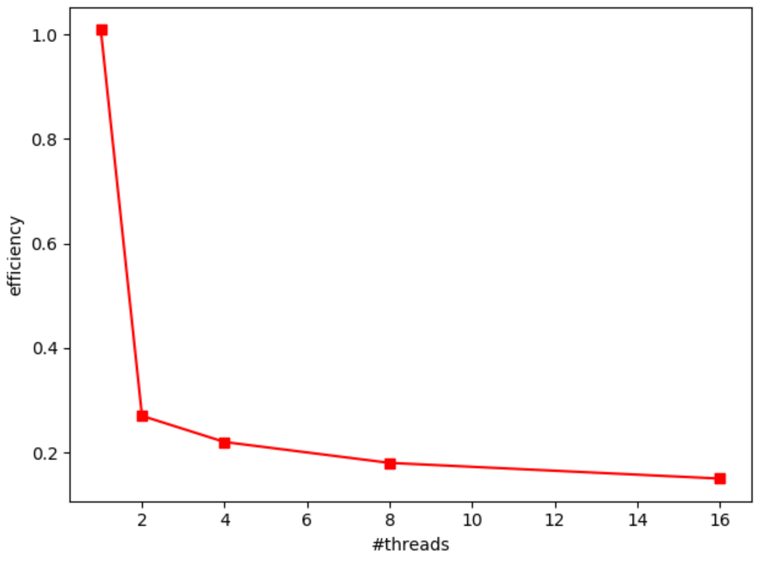


- The strong and weak scaling plots that show the efficiency of our optimized OpenMP implementation

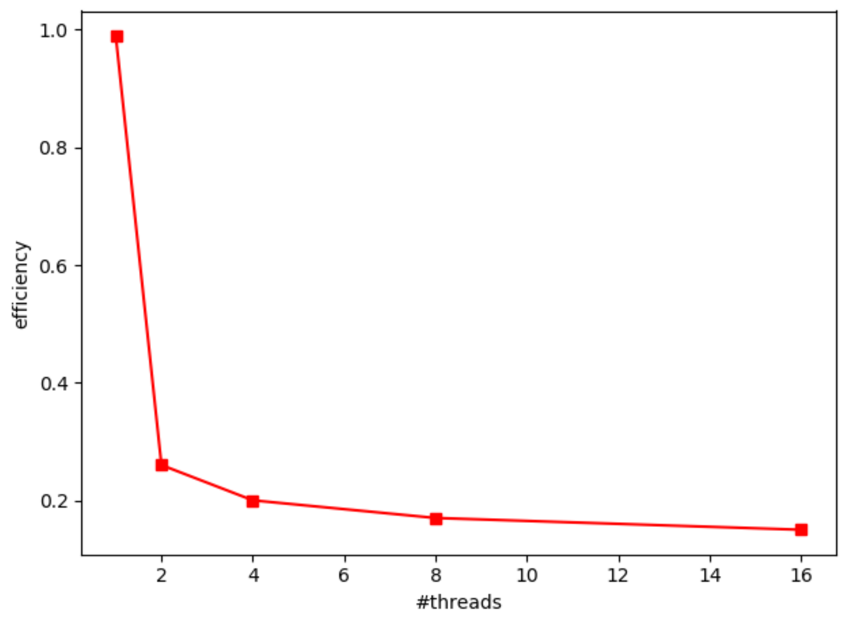
Strong scaling plot of efficiency for the small dataset



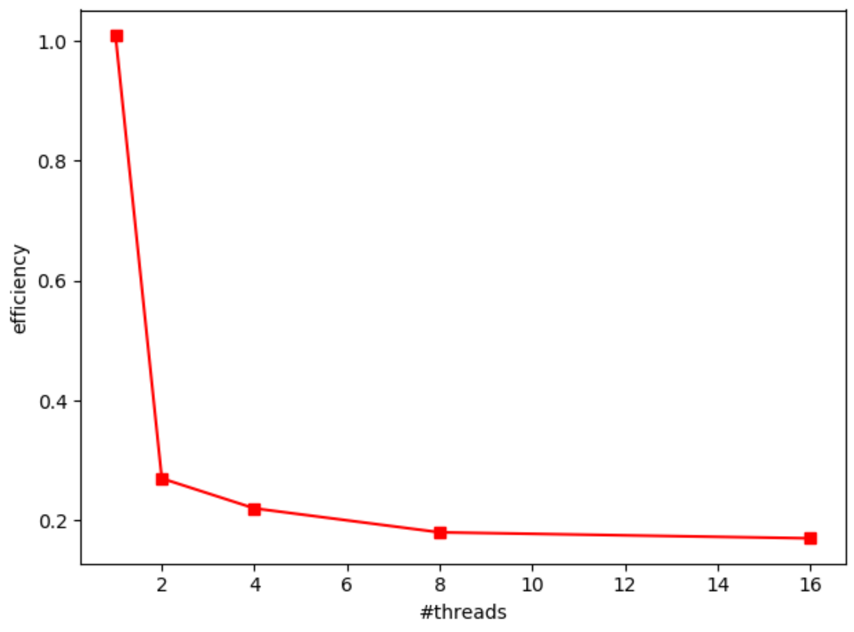
Strong scaling plot of efficiency for the large dataset



Weak scaling plot of efficiency for the small dataset



Weak scaling plot of efficiency for the large dataset



Analysis:

For the strong scaling plots for running time, we can see that the running time increases at **thread number = 2**, then decreases as the number of threads increases. We see the same pattern with both the small dataset and the large dataset.

For the weak scaling plots for running time, we see that the running time increases as the particle size grow proportionally with the number of threads, but the slope of each line segment is getting smaller. In other words, the running time in the plots increases, but with a decreasing rate of change. We see the same pattern with both the small dataset and the large dataset.

The strong scaling speedup plots shows that the speedup initially decreases at **thread number = 2**, then increases afterward. The pattern is the same with both the small dataset and the large dataset.

For the weak and strong scaling efficiency plots, we see that the efficiency gradually decreases as the number of threads goes up. As the number of threads increases, we see a faster drop in efficiency at the first and a slower drop in efficiency later. The pattern is the same with both the small dataset and the large dataset.

There are two major observations we made in our experiments. For the weak scaling plots, even though work/processor stays the same for each data point in the plots, we still see performance drops as the number of threads increases. This can be explained by the overhead of opening extra threads and the overhead caused by the synchronization of more threads.

For the strong scaling plots, especially the speedup plots. We see a performance drop at **thread number = 2**, then increases afterward. We think the reason for the performance drop at the beginning is due to the following possible reasons:

1. Interprocessor communication: The particles are randomly distributed in the whole map. The less number of threads, the more particles that each thread needs to handle which causes more communication. This is parallel overhead.
2. Load imbalance: Our design can't ensure load balance because some particles may not have any other particles in nearby grids.

Our plots, especially the speedup plots, also show better performance with bigger datasets. This is expected as the **Amdahl effect**, for the same number of processors, the speedup is usually an increasing function of the problem size.

Suppose p is the number of threads and idealized p-time speedup to be the speedup equaled to the current number of threads. If we compare with the idealized p-time speedup, our openmp implementation has slightly less(about 96%) idealized p-time speedup when p = 1, around 20% - 30% idealized p-time speedup when p is 2 and 4, less than 20% idealized p-time speedup when p increases to 8 and 16. Basically, even though our implementation's speedup is increasing when #threads >= 2, when we compare with the idealized p-time speedup, the ratio of this speedup and idealized p-time speedup is decreasing as p increases.

This difference between our speedup and the idealized p-time speedup can be explained by: context switching overhead, communication overhead caused by the critical section we created with C++ "atomic" and load imbalance. It is possible to achieve a better speedup. One possible approach is to use better scheduling strategies. Better scheduling will mitigate the load imbalance and increase the speedup.

Time and Locking Strategy

In our program, the time mostly goes in retrieving the data from memory to cache because we used the linked list data structure. However, because the mutex lock is expensive to use when the critical section is very small just like our case, we choose to use the c++

atomic type which provides spinlock. We implement our Lock-free Linked List which improves the overall runtime from 15 seconds to 11 seconds in 160000 particles simulation.