# CSC726
# Particle Simulation Part 1&2

**Ziqin Chen, Ziji Zhang**

## 1. Complexity of Naive Way for serial.cpp

According to the "autograder.cpp", the naive way to simulate particles in serial has complexity of $O(n^2)$.

| Number of particles | Time(sec) |
|---|---|
| 100 | 0.026495 |
| 200 | 0.098828 |
| 400 | 0.379047 |
| 800 | 1.49779 |
| 1600 | 5.95669 |
| 3200 | 23.9752 |

Serial code is $O(n^{slope})$.

Slope estimates are : 1.899200, 1.939385, 1.982387, 1.991676, 2.008960.

Slope estimate for line fit is: 1.966593.

The estimate complexity is $O(n^2)$ according to those records, which is common for serial algorithm but not satisfying.

## 2. Complexity of our improved serial.cpp

| Number of particles | Time(sec) |
|---|---|
| 100 | 0.010995 |
| 200 | 0.023029 |
| 400 | 0.050661 |
| 800 | 0.108105 |
| 1600 | 0.234276 |

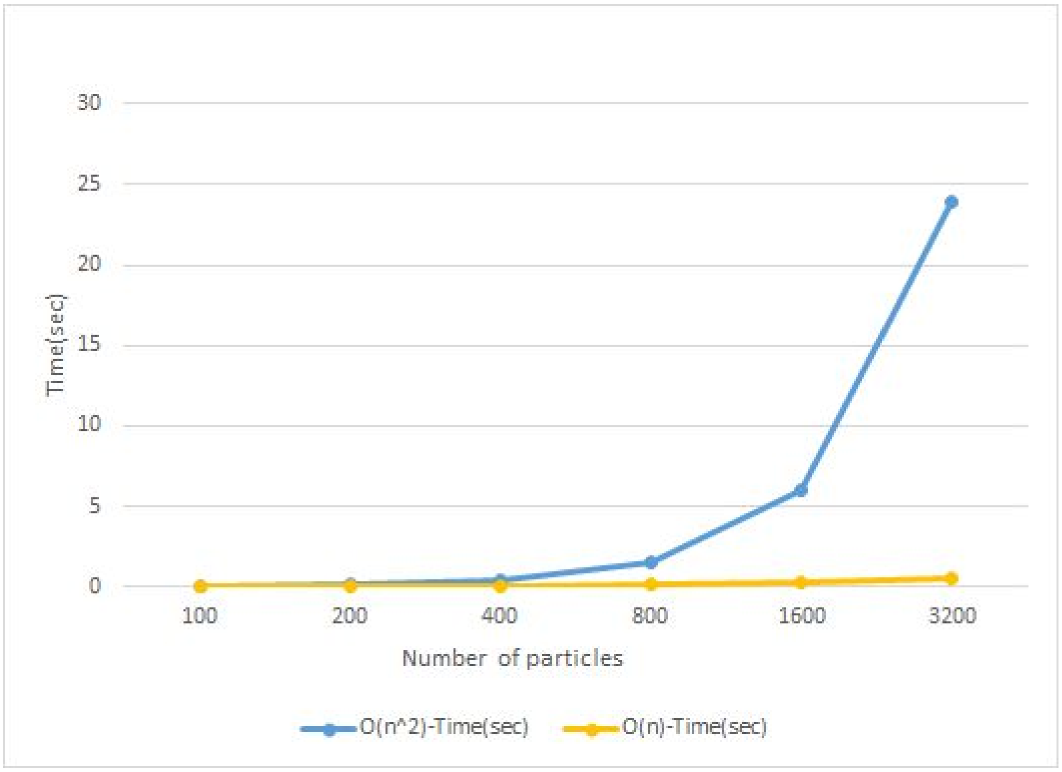
| | |
|---|---|
| 3200 | 0.558938 |

Serial code is $O(n^{slope})$.

Slope estimates are : 1.066604, 1.137424, 1.093486, 1.115776, 1.254479

Slope estimate for line fit is: 1.127782

The estimate complexity is $O(n)$ according to those records.



In the native version of serial.cpp, for each particle in each time step, we need to calculate the forces on that particle and check for the interaction radius from all other particles in the system. But most comparisons are outside interaction range, so it's a good idea to divide the particles into several equal sized bins and then we will only check particles that belongs to its neighbor bins.

In our improved serial.cpp, we create a 2D vector for the bins and in every time step, we will push all the particles into its corresponding bins based on the calculation of $x$ and $y$. And then we compare all the particles in neighbor bins (including this bin) with this specific particle itself and call **apply_force** function. Before entering the next time step, we have to resize the size of each inner vector back to be 0 in order to update the bins for particles after applying force.

By checking only neighboring bins reduces the complexity from $O(n)$ for each particle to $O(9d)$ where $d$ is the average number of particles in each bin. Thus , overall complexity will be reduced from $O(n^2)$ to $O(9dn)$ . Since density in this program is uniform and constant, we can consider this $d$ to be a tiny constant so that our final complexity is $O(n)$ which matches what we observed from the autograder.cpp.

## 3. Speedup and efficiency of Naive Ways for openmp.cpp

| Number of particles | Number of Threads | Time(sec) |
|---|---|---|
| 500 | 1 | 0.715909 |
| 500 | 2 | 0.822034 |
| 500 | 4 | 0.485354 |
| 500 | 6 | 0.403393 |
| 500 | 8 | 0.378766 |
| 500 | 11 | 0.401185 |
| 500 | 22 | 0.382432 |
| 500 | 26 | 0.380846 |
| 500 | 30 | 0.376698 |
| 500 | 32 | 0.400941 |
| 1000 | 2 | 3.1057 |
| 2000 | 4 | 4.02383 |
| 3000 | 6 | 5.41977 |
| 4000 | 8 | 7.11327 |
| 8000 | 16 | 17.0887 |
| 16000 | 32 | 32.4907 |

Strong scaling estimates are :

0.81 0.71 1.20 1.44 1.54 1.45 1.52 1.53 (speedup)
0.81 0.35 0.30 0.24 0.19 0.13 0.07 0.06 (efficiency) for
1 2 4 6 8 11 22 26 (threads/processors)

Average strong scaling efficiency: 0.27

Weak scaling estimates are :

0.81 1.54 1.45 0.19 0.14 0.11 0.08 0.03 (efficiency) for
1 2 4 6 8 11 22 26 (threads/processors)

Average weak scaling efficiency: 0.55

## 4. Speedup and efficiency of Our openmp.cpp

| Number of particles | Number of Threads | Time(no locker) |
|---|---|---|
| 500 | 1 | 0.058979 |
| 500 | 2 | 0.061174 |
| 500 | 4 | 0.03953 |
| 500 | 6 | 0.036109 |
| 500 | 8 | 0.034584 |
| 500 | 11 | 0.034749 |
| 500 | 22 | 0.032989 |
| 500 | 26 | 0.07351 |
| 500 | 30 | 0.059891 |
| 500 | 32 | 0.126884 |
| 1000 | 2 | 0.116542 |
| 2000 | 4 | 0.12243 |
| 3000 | 6 | 0.172327 |
| 4000 | 8 | 0.201554 |
| 8000 | 16 | 0.349213 |
| 16000 | 32 | 1.65673 |

Strong scaling estimates are :

```
    0.96   1.49   1.63   1.71   1.70   1.79   0.80    (speedup)
    0.96   0.37   0.27   0.21   0.15   0.08   0.03    (efficiency)   for
     2      4      6      8      11     22     26     threads/processors
```

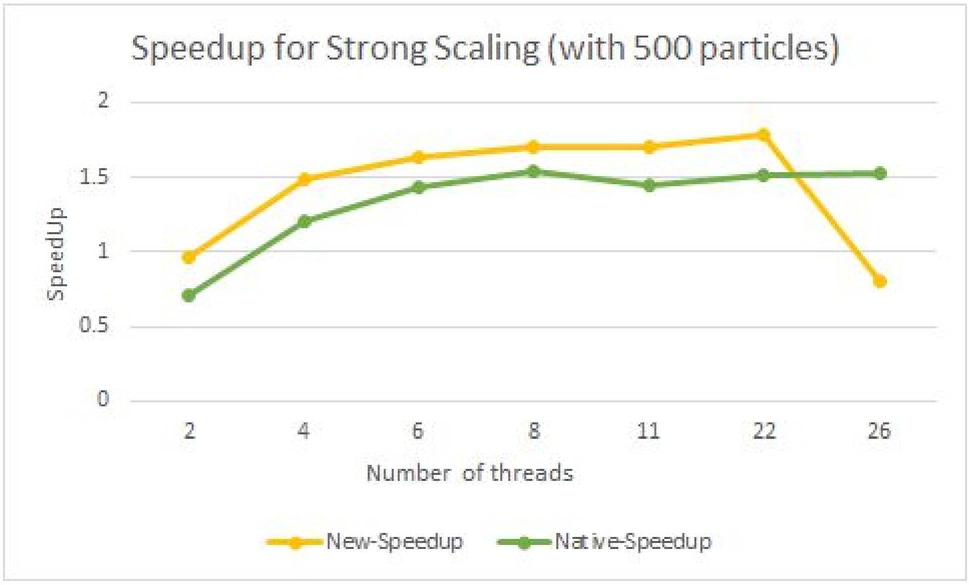
Average strong scaling efficiency:  0.30

Weak scaling estimates are :

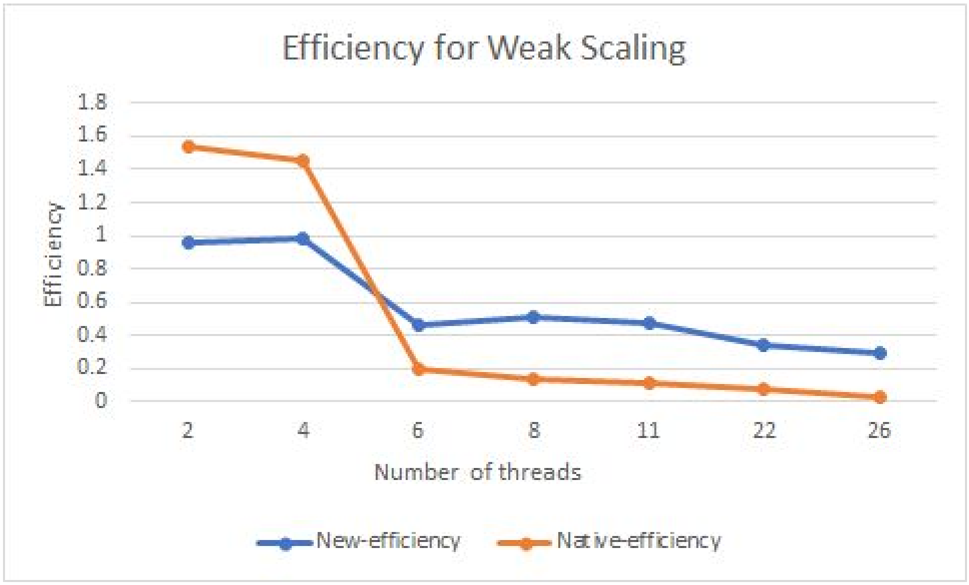
```
   0.96   0.98   0.46   0.51   0.48   0.34   0.29  (efficiency)   for
     2      4      6      8      11     22     26   threads/processors
```

Average weak scaling efficiency:    0.58

## 5. Analysis



From the graph of strong scaling, we can see that the improved openmp.cpp will give us a better speedup from 2 processors to around 22 processors. After 22 processors, the improved openmp.cpp's speedup decreased sharply, and after some points it becomes lower than the naive openmp.cpp's speedup. We assume this happens because the improved openmp.cpp will have larger cost to spawn threads compared to the cost saved by parallelization before the naive openmp.cpp will do. This point varies when the problem size changes.



From the graph of weak scaling, we can see that the improved openmp.cpp will give us a better speedup from after using 6 processors. Before we are using 6 processors, the naive openmp.cpp has a better efficiency. We assume this happens because the problem size we used for small number of processors are small, so the cost to spawn threads and parallelize will be larger than what the parallelization saved.

## 6. openmp.cpp (old version without lockers)

We parallized our serial.cpp whose complexity is $O(n)$ in a shared memory model by using OpenMP.

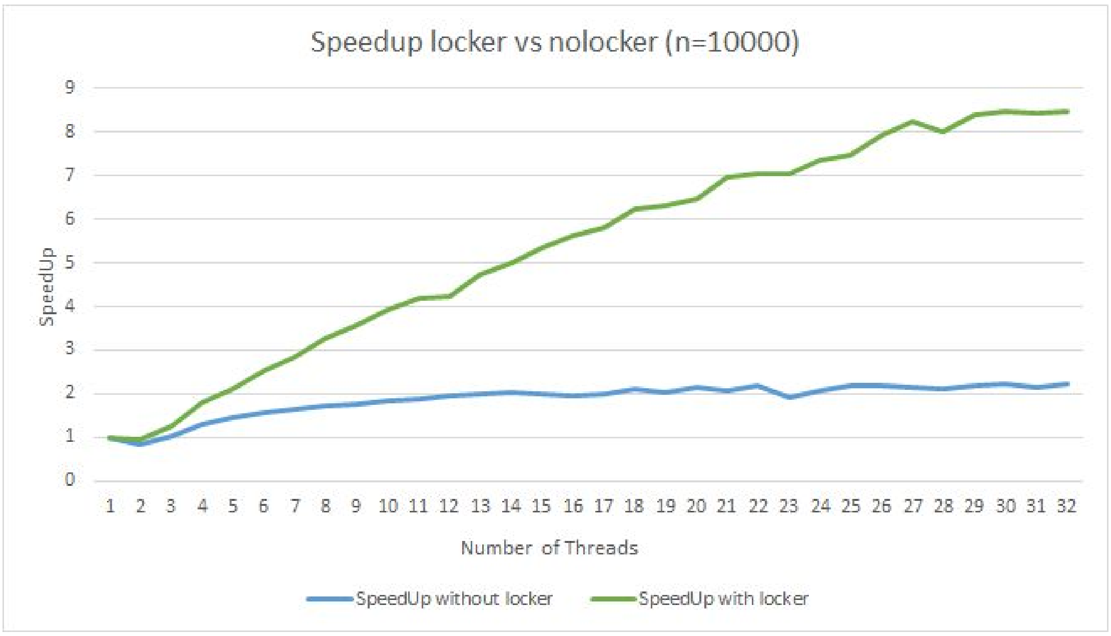
Firstly, we tried to parallelize the outermost loop for Nsteps, but we failed to deal with the shared variables **dmin**, **davg**, and **navg**, and it couldn't compile successfully. We tried several ways such as making **dmin** private, and using a reduction pragma to sum up davg and **navg**, but still failed. Thus, we commented out this attempt.

Secondly, we tried to parallelize the **for**-loop which is to push_back each particle into its corresponding bin. The **puch_back** function can't be parallelized because if two threads are doing **push_back** to the same vector, a segmentation fault will take place. Thus, we only parallelized the first two lines of the **for**-loop (which is calculating the row and column indices of corresponding bins), and make the **push_back** function a critical section. However, we found the simulation time increased when we parallelized this **for**-loop. The reason should be that we only parallelize the loop by a factor of 2, but the cost to spawn the threads and critical section exceeds the cost we have saved. Thus, we commented this attempt out.

Then, we parallelized the second nested **for**-loop simply by adding pragma "omp parallel for". We also did the same thing for **move** particles function and **resize** the bin vector. We make the master thread to do some final computation in the end.

## 7. openmp.cpp with lockers!

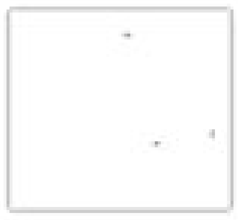
We use **omp_lock** to optimize the parallelization for our **push_back** function. By using this locker, we can push different particles which belongs to different bins into its corresponding bin simultaneously instead of pushing all of the particles serially. We compared the speed-up of our new openmp.cpp with that of the old version and found if the problem size is not big enough, speedup without lockers looks better than the one has lockers. The reason might be that cost for using lockers exceeds the cost we have saved. So we plot the speedup with problem size 10000 which has a great optimization.



## 8. MPI

For mpi.cpp, thread 0 has all the particles, and then we broadcast all the particles to the other threads. Thus for each thread, they will only call **apply_force** function on its data chunk which means each thread focuses on its particles at the same time. Then we tried to update particles' new positions after we move it before entering next time step. We tested with gathering all the particles into thread 0 and broadcasting the updated particles to the other threads or allgather but we were not sure about whether it would overwrite data by using these mpi functions. Then, we tried to create a temp variable to store particles and unfortunately it doesn't work.

We visualized our output by using 2 threads and 3 particles. In the first output we can only see the movement of particles we assigned to thread 0 which has 2 particles.



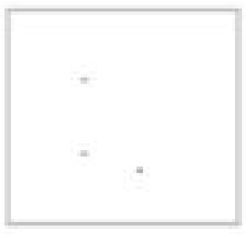
Then we changed a little bit of this part of code from :

```
FILE *fsave = savename && rank == 0 ? fopen( savename, "w" )
```

to be

```
FILE *fsave = savename && rank == 1 ? fopen( savename, "w" )
```

Thus the output will show the movement of the other part of particles. In the second output we can only see the movement of particles we assigned to thread 0 which has 1 particle.



We also print the simulation time out for each thread, P0 usually has the largest simulation time since it has more work to do.

```
Case 1:
P0
n = 3, simulation time = 0.003061 seconds
P1
n = 3, simulation time = 0.001145 seconds

Case 2:
P2
n = 100, simulation time = 0.006272 seconds
P3
n = 100, simulation time = 0.006257 seconds
P0
```

```
n = 100, simulation time = 0.006653 seconds
P1
n = 100, simulation time = 0.006603 seconds
```

**(We will try to find a solution! )**