HW2: Parallelizing a Particle Simulation

Nuochen Lyu

Xiaoyun Zhao

Part I. Serial and OpenMP

1. Introduction

In this particle simulation, the naive implementation obtains the time complexity of $O(n^2)$ by looping over each particle in the canvas. The goal of our serial code is to achieve $O(n)$ by implementing the Bin structure and only search for particles in 9 neighboring bins (including the current bin itself) during interactions. After the serial code we implemented the OpenMP solution which takes the advantage of shared memory model and aims to parallelize the process and achieve $O(n / p)$ with p processors.

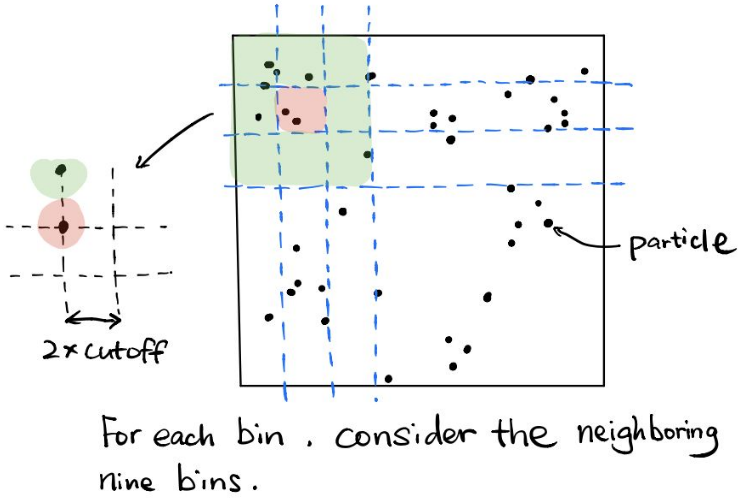
2. Serial Implementation

2.1 Data Structure

To achieve O(n) for the particle simulation, we implemented the serial code using an underlying “bin” structure. In the bin class, we defined nei_id to point to all neighboring bins (including the bin itself), par_id to point to a list of particles that belongs to the current bin, num_par and num_nei to keep track of the total number of particles and neighboring bins. The structure was implemented as follows:

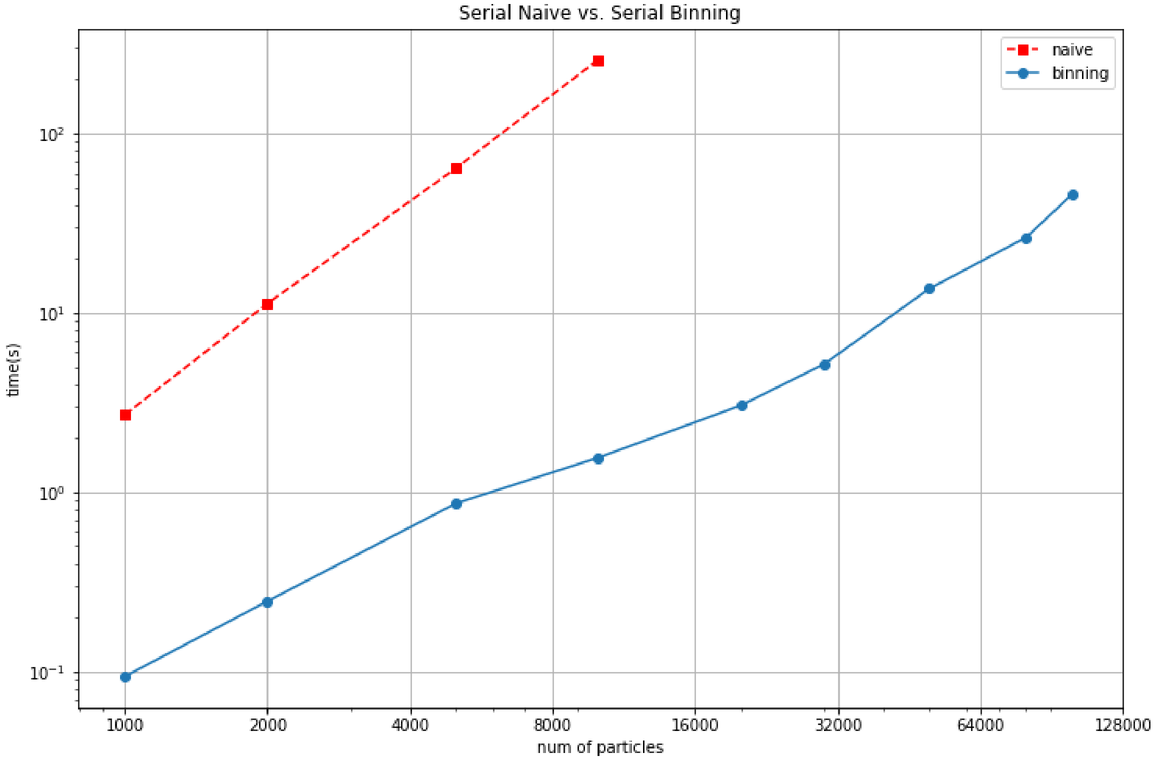
```
class bin{
public:
    int num_par, num_nei;   //counter
    int * nei_id;           //neighboring bins
    int * par_id;           //particles in the bins

    bin(){
        num_nei = num_par = 0;
        nei_id = new int[9];
        par_id = new int[particle_num];
    }
};
```

For each particle, other than checking all the rest of the particles, we only examine the particles in the same bin and 8 neighboring bins. Thus the time complexity can be reduced from to $O(n)$ on average case. Below is an illustration:



2.2 Results

Compare to the naive implementation, the serial code significantly reduce the time complexity. The result is shown in the below log-log graph:



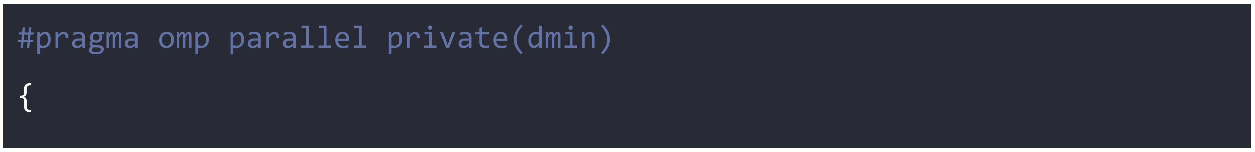
3. OpenMP

3.1 Implementation Discussion

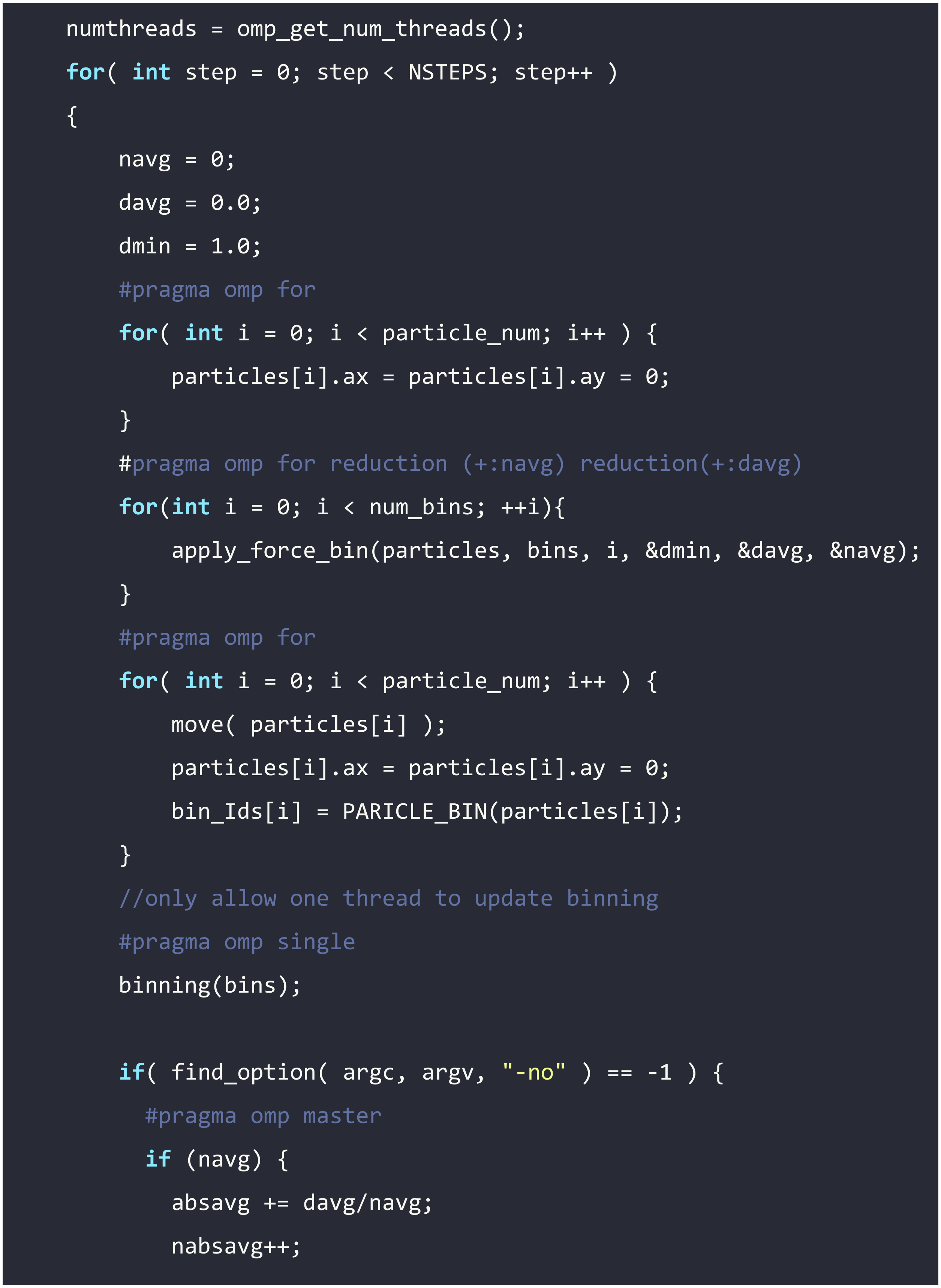
We attempted different approaches to achieve synchronization in OpenMP. In this discussion, we are going to talk about two implementations and compare the results accordingly.

3.1.1 Single Worksharing Construct

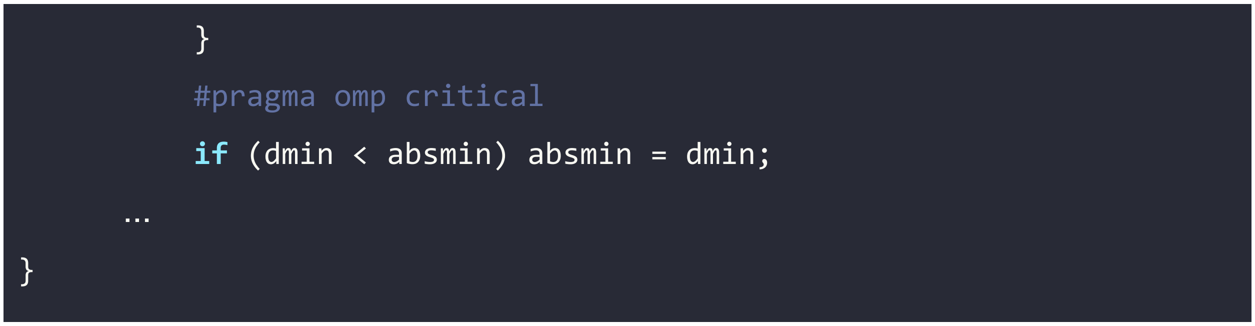
To avoid false sharing condition we implemented the single work-sharing construct to allow only a single thread to update particle positions during the binning process. The code is as follows:



```
#pragma omp parallel private(dmin)
{
```

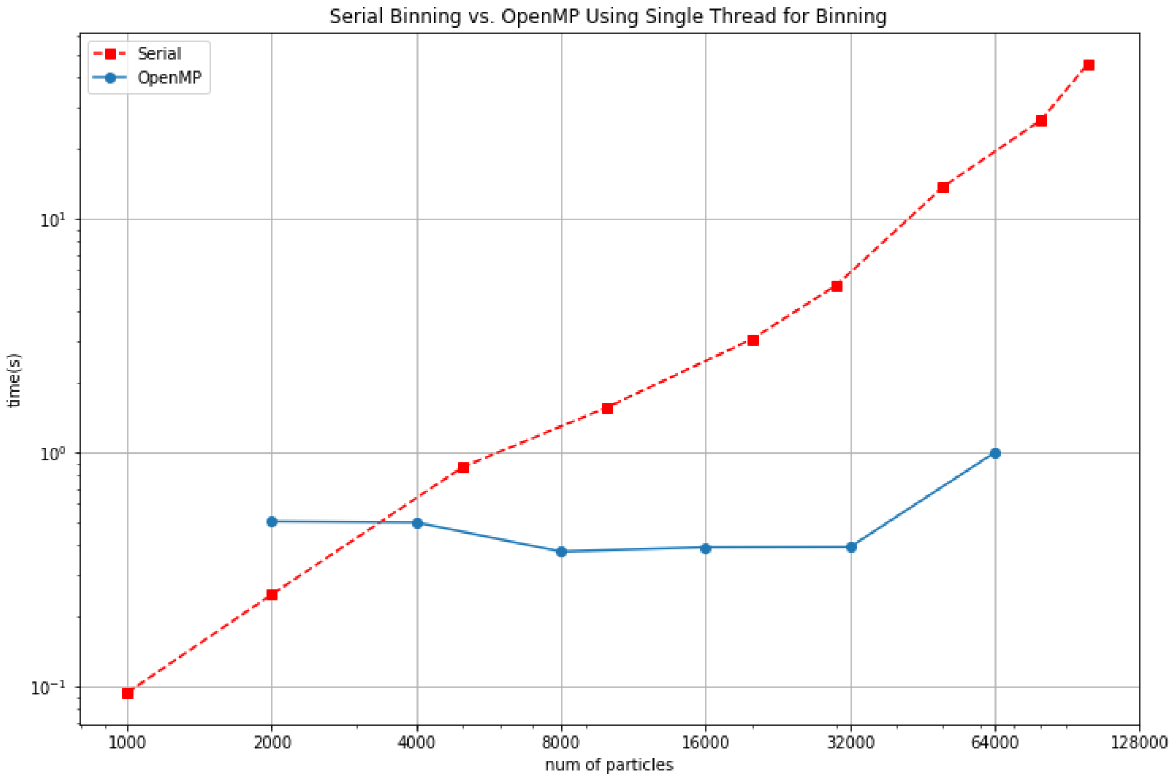


```
numthreads = omp_get_num_threads();
for( int step = 0; step < NSTEPS; step++ )
{
    navg = 0;
    davg = 0.0;
    dmin = 1.0;
    #pragma omp for
    for( int i = 0; i < particle_num; i++ ) {
        particles[i].ax = particles[i].ay = 0;
    }
    #pragma omp for reduction (+:navg) reduction(+:davg)
    for(int i = 0; i < num_bins; ++i){
        apply_force_bin(particles, bins, i, &dmin, &davg, &navg);
    }
    #pragma omp for
    for( int i = 0; i < particle_num; i++ ) {
        move( particles[i] );
        particles[i].ax = particles[i].ay = 0;
        bin_Ids[i] = PARICLE_BIN(particles[i]);
    }
    //only allow one thread to update binning
    #pragma omp single
    binning(bins);

    if( find_option( argc, argv, "-no" ) == -1 ) {
      #pragma omp master
      if (navg) {
        absavg += davg/navg;
        nabsavg++;
```

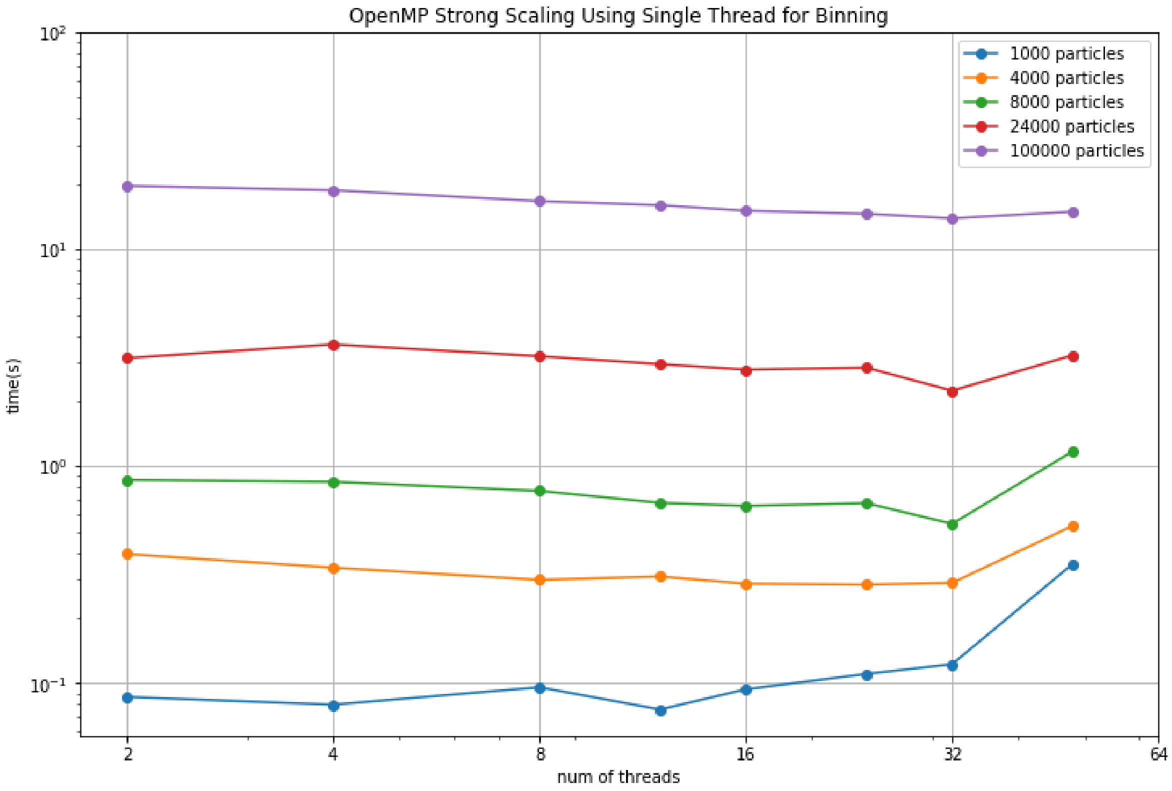


```
            }
            #pragma omp critical
            if (dmin < absmin) absmin = dmin;
        …
}
```

This implementation allows all other sections to be parallelized, while maintain the integrity of the shared memory by all threads. Compare to the serial implementation, the OpenMP solution does accelerate the simulation process through parallelization, which can be shown in the below graph:

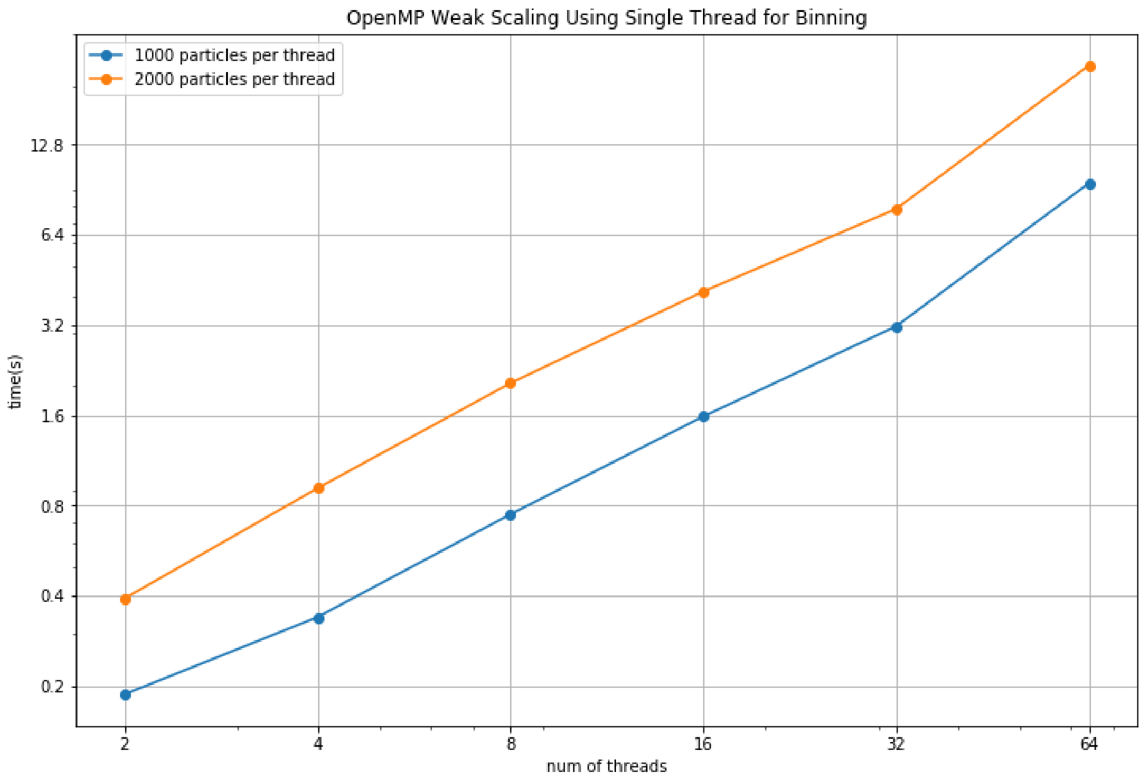


However, the instruction of only allowing a single thread to update binning also became the bottleneck of the parallelization. We can roughly observe that the changing rate of the OpenMP's function remains relatively constant while the number of threads is actually increasing. To confirm this behavior we ran more jobs and plotted the graph below:



The above log-log graph shows the strong scaling results with OpenMP implementation ranging for 1000, 4000, 8000, 24000, and 100000 particles respectively. We can observe that the running time did not get better as the num of threads increases, which proved that there’s a bottleneck affecting the performance of parallelization.

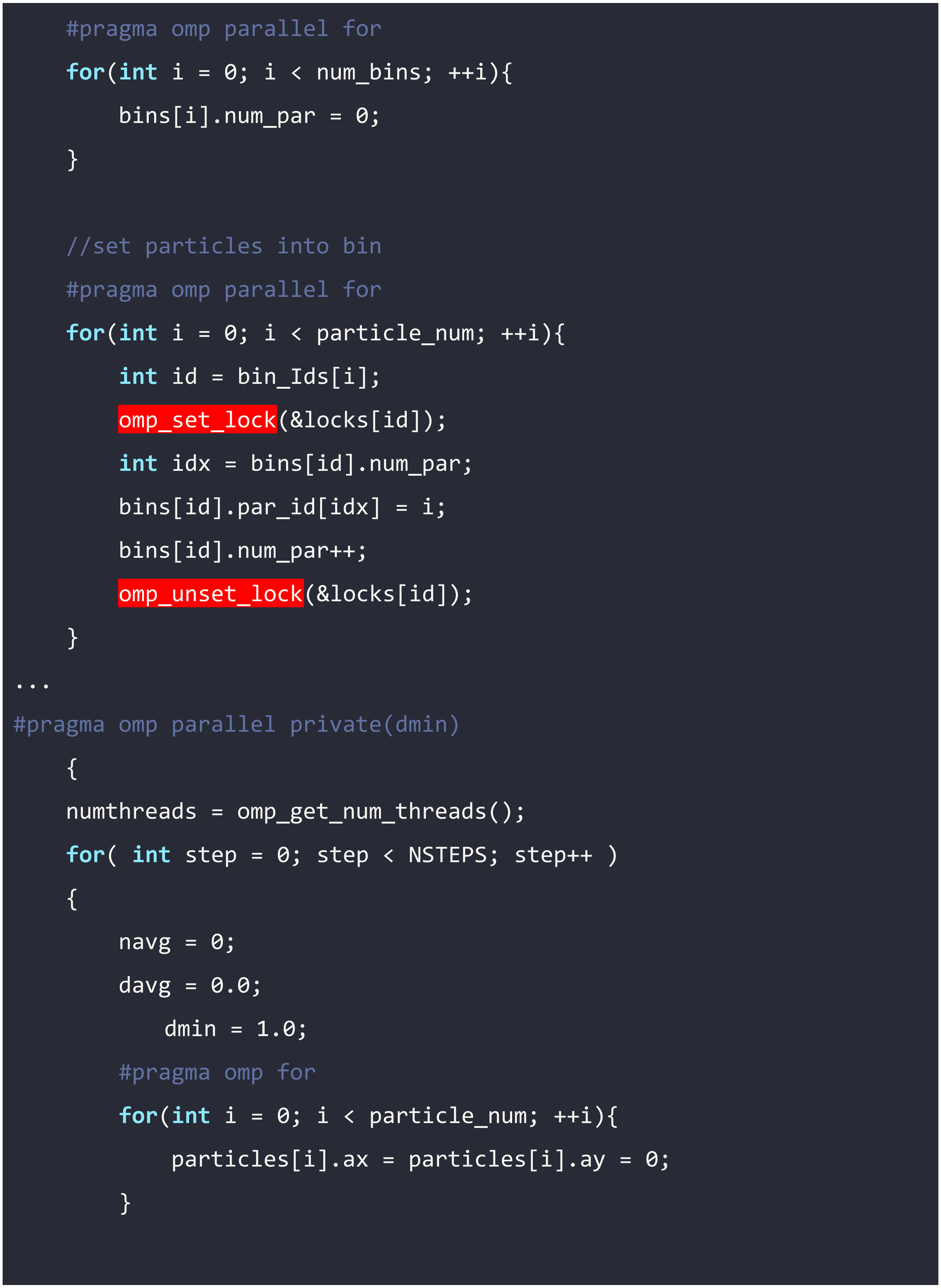
Another log-log graph for weak scaling using a single thread for binning is shown below.



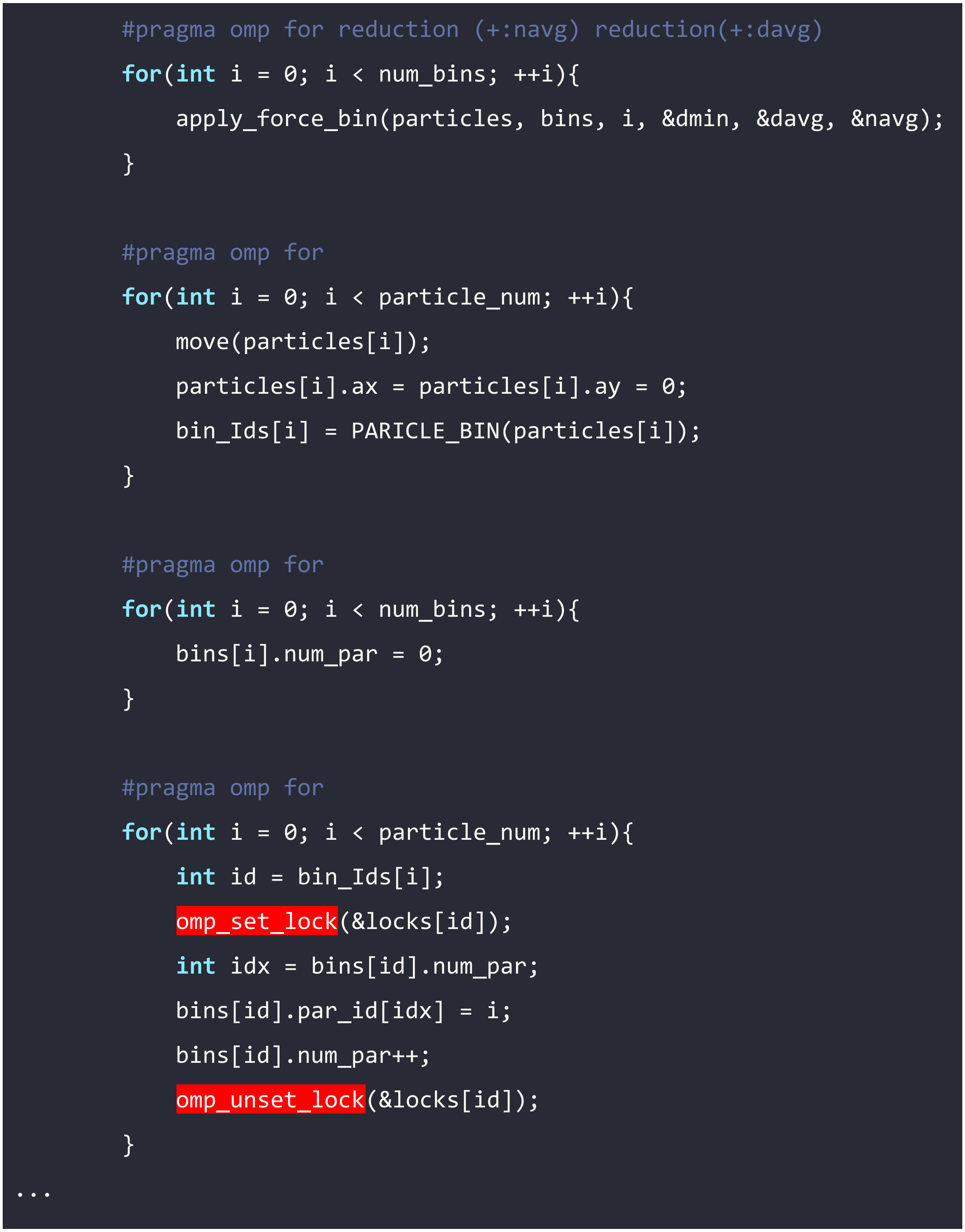
3.1.2 Using omp_set_lock

We realized that although single work-sharing construct avoids race condition, it slows down the parallel effort to a great extent. Therefore we implemented low-level synchronization solution `omp_set_lock` and `omp_unset_lock`, which assigned a lock to each bin, and locks the bin's access while a certain thread is updating particle interactions in that bin. The code is shown below:

```
...
//map the bins mack to particle
    omp_lock_t * locks = new omp_lock_t[num_bins];
    for(int i = 0; i < num_bins; ++i)
        omp_init_lock(&locks[i]);
```

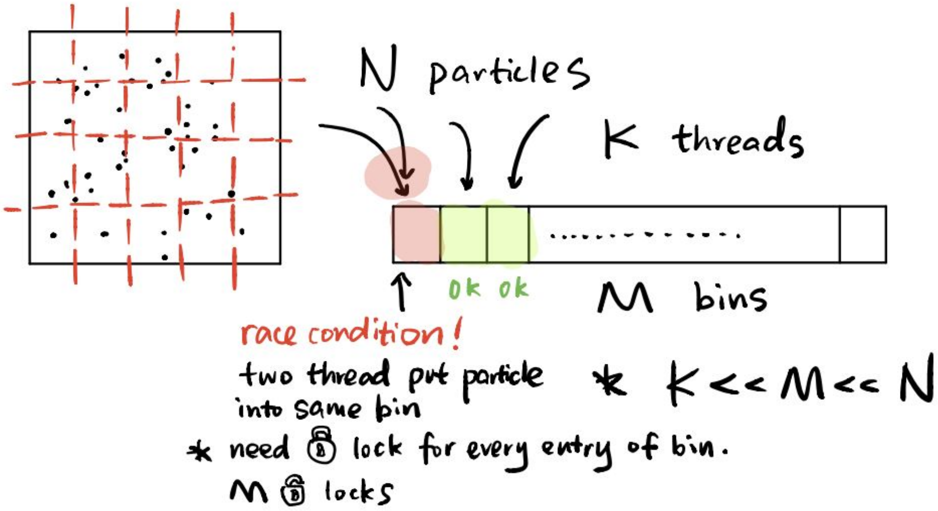


```
    #pragma omp parallel for
    for(int i = 0; i < num_bins; ++i){
        bins[i].num_par = 0;
    }

    //set particles into bin
    #pragma omp parallel for
    for(int i = 0; i < particle_num; ++i){
        int id = bin_Ids[i];
        omp_set_lock(&locks[id]);
        int idx = bins[id].num_par;
        bins[id].par_id[idx] = i;
        bins[id].num_par++;
        omp_unset_lock(&locks[id]);
    }
...
#pragma omp parallel private(dmin)
    {
    numthreads = omp_get_num_threads();
    for( int step = 0; step < NSTEPS; step++ )
    {
        navg = 0;
        davg = 0.0;
           dmin = 1.0;
        #pragma omp for
        for(int i = 0; i < particle_num; ++i){
            particles[i].ax = particles[i].ay = 0;
        }
```

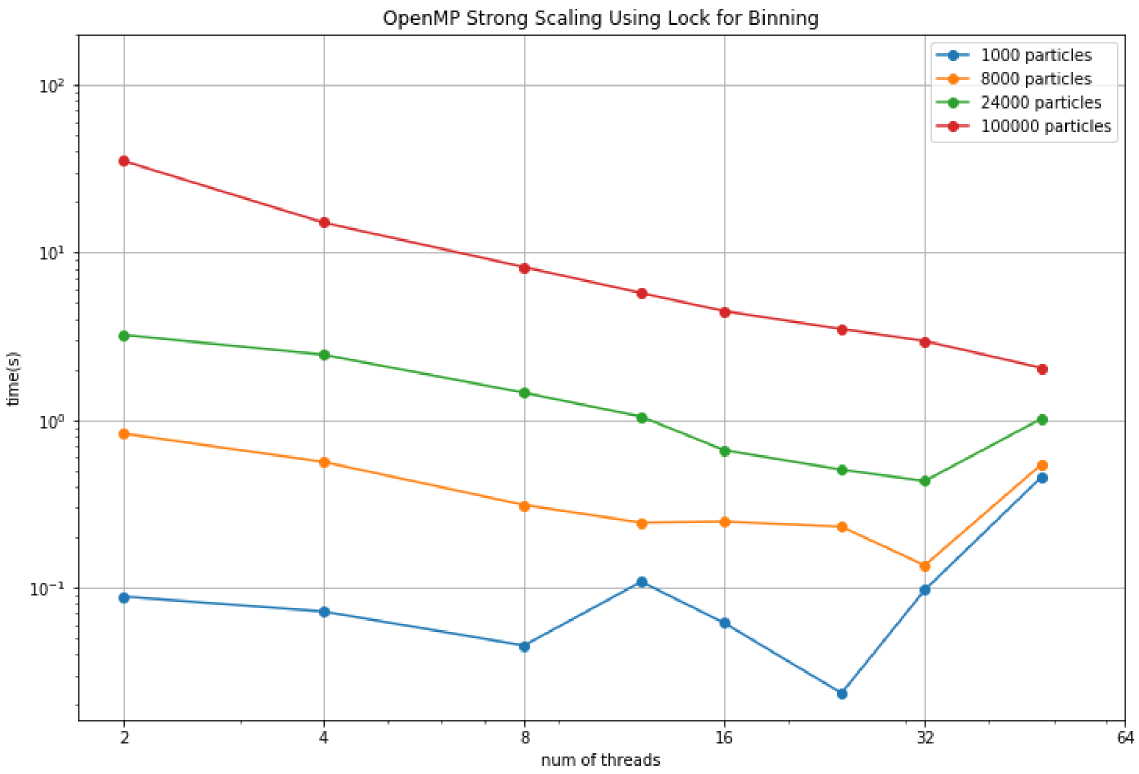


```
        #pragma omp for reduction (+:navg) reduction(+:davg)
        for(int i = 0; i < num_bins; ++i){
            apply_force_bin(particles, bins, i, &dmin, &davg, &navg);
        }

        #pragma omp for
        for(int i = 0; i < particle_num; ++i){
            move(particles[i]);
            particles[i].ax = particles[i].ay = 0;
            bin_Ids[i] = PARICLE_BIN(particles[i]);
        }

        #pragma omp for
        for(int i = 0; i < num_bins; ++i){
            bins[i].num_par = 0;
        }

        #pragma omp for
        for(int i = 0; i < particle_num; ++i){
            int id = bin_Ids[i];
            omp_set_lock(&locks[id]);
            int idx = bins[id].num_par;
            bins[id].par_id[idx] = i;
            bins[id].num_par++;
            omp_unset_lock(&locks[id]);
        }
...
```

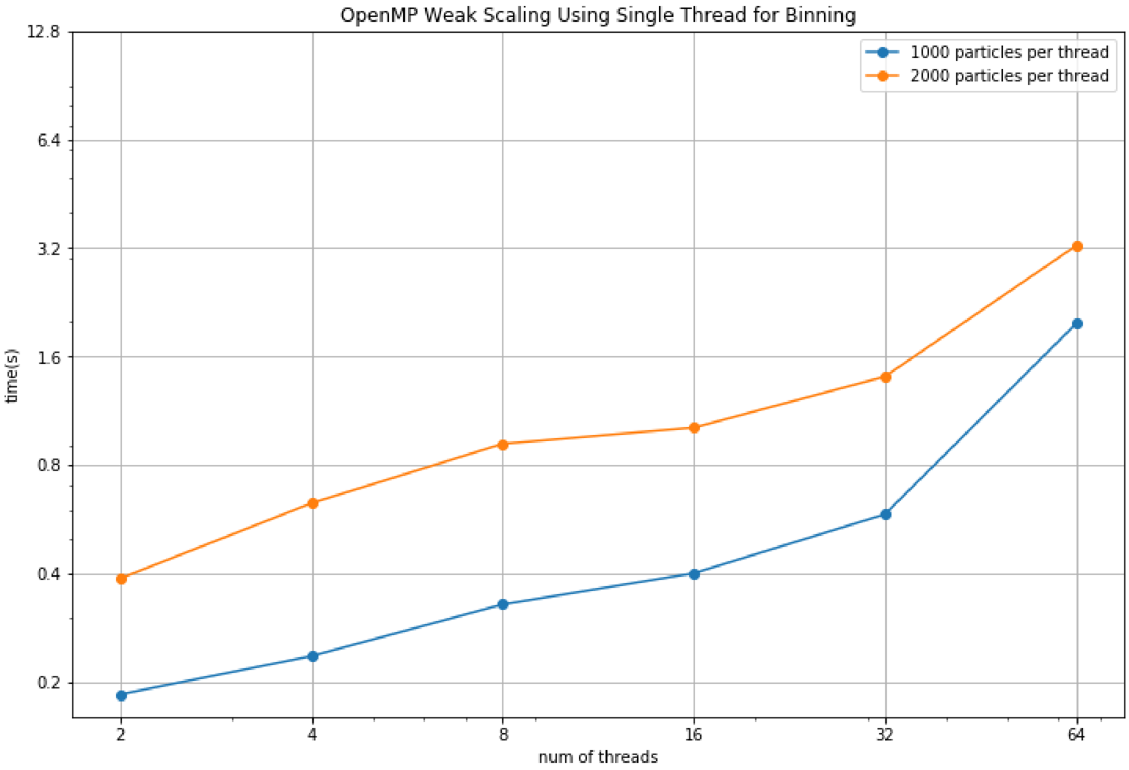
More detailed explanation is shown in the below illustration:


N particles
K threads
ok ok
M bins
race condition!
two thread put particle into same bin
* K << M << N
* need 🔒 lock for every entry of bin.
M 🔒 locks

This implementation shows great improvement in parallelizing with an increasing number of threads. We can observe that this implementation approached the idealized p-times speedup which brought the run time down to approximately $O(n / p)$ with p processors. The results for strong scaling can be found below:



Compared to the previous implementation using a single work-sharing construct, the overall run time for weak scaling also drops significantly (from 19.7s to 3.2s for 128000 particles with 64 threads). The results for weak scaling is shown below:



3. Conclusion

We implemented serial code to reduce the time complexity from $O(n^2)$ to $O(n)$ through the bin data structure, then implemented OpenMP solutions to parallel the process under shared memory model and achieved p-times speedup with p processors. Some tips during the implementation of OpenMP solutions:

- Thread creation overhead
- Debugging data race condition
- The use of Reduction, thread-private variables
- Inefficient use of barrier / critical synchronization strategies