# Parallel Particle-In-Cell Method

**Yuanxin Cao**
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213
yuanxinc@andrew.cmu.edu

**Lan Lou**
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213
lanlou@andrew.cmu.edu

## Summary

This project aims to parallelize the Particle-in-Cell (PIC) algorithm in a 3D space using OpenMP and MPI libraries. It explores two OpenMP parallelization strategies, including parallelizing by particles and grids, and three MPI parallelization strategies, including parallelizing by particles, grids, and a hybrid approach. Through careful analysis and experimentation, the project optimizes performance for varying grid sizes and particle counts, achieving significant speedup while minimizing communication overhead.

## 1 Background

### 1.1 PIC Algorithm

The objective of this project is to implement the Particle-in-Cell (PIC) algorithm [1] in a 3D space and parallelize it using the OpenMP [2] and MPI [3] libraries. In the PIC algorithm, the plasma is represented by a large number of discrete charged particles, typically electrons and ions. Each particle carries attributes of position, velocity, and charge. These particles are used to represent the behavior of the entire plasma system. By tracking the trajectories of individual particles, the collective behavior of the plasma can be simulated.

The computational domain is divided into a grid, as fig. 1 shows (though it is shown in a 2D way while we adopt a 3D grid in our code). Each grid point stores information about the value of the electric field at the very point. The charged particles are shown in fig. 1 as the black dots. They scatter randomly across the grid, and the motion of the particles is governed by the Coulomb force, which accounts for the forces experienced by particles due to electronic fields.

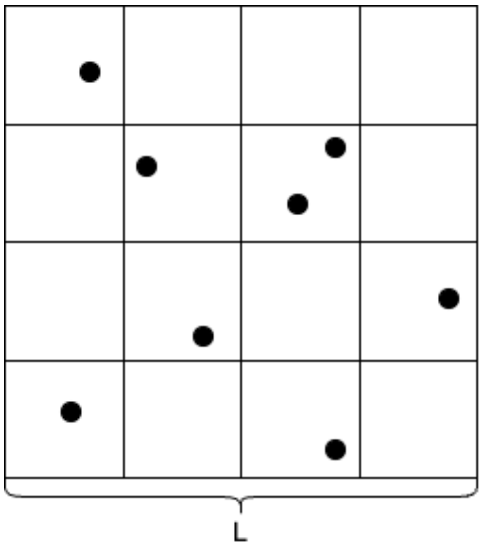


Figure 1: Basic physics setting (2D)

The electronic field at an arbitrary point is interpolated based on the electronic fields at the surrounding grid points. We can then calculate the Coulomb force for a given charged particle based on the interpolated electronic field.

The simulation progresses through discrete time steps, with particle positions and velocities updated at each time step. The program accepts some parameters as inputs, including the size of the grid, the number of charged particles, the number of iterations to step through, and the way of particle position initialization. The program will output the final position of the charged particles, indicating their motion in the electronic field during the whole process.

We can divide the main loop which can be sped up into 2 parts [4]:

1. Update the positions and velocities of all particles according to the interpolated values derived from the electronic field grid at their current positions.
2. Update each grid point in the electronic field based on all particles' charges and positions.

The first part is totally parallelizable since it only entails pure computation, without interleaved global memory access across threads.

For the second part, we will explore two algorithms for updating the grid point. In the first algorithm, we calculate the electronic field for each grid point by visiting all points in the environment, just as the pseudo-code below shows. The time complexity for this algorithm should be $O(L^3 * P)$, where $L$ is the number of grid points on one side, and $P$ is the number of charged particles. This part can be parallelized if we partition the work by the grid points. However, each worker needs a whole copy of the charged particles to compute the contribution of each charged particle and add them up.

This algorithm is more amenable to the setting where we have a large number of grid points and a relatively small number of particles.

```
for grid_point in grid_points {
    electronic_field = 0;
    for particle in particles {
        electronic_field += computeElectronicField(grid_point, particle);
    }
    Egrid[grid_point] = electronic_field;
}
```

The second algorithm is to calculate the contribution of the electronic field for each charged particle and then accumulate all updates for each grid point, just as the pseudo-code below shows. The time complexity should be $O(P)$, since the number of the surrounding grid points is a constant for a given particle ($8$ in our 3D setting). This could be parallelized by charged particles, but since the positions of the charged particles might overlap, it is possible that two workers update the same grid point, and the update part thus needs to be atomic.

```
for particle in particles {
    surrounding_grid_points = getSurroundingGridPoints(particle);
    for grid_point in surrounding_grid_points {
        Egrid[grid_point] += computeElectronicField(grid_point, particle);
    }
}
```

In the analysis below, if not specially mentioned, we will use the first algorithm as the base algorithm and do the optimizations on top of that one.

## 2 Approach

We change the code from Kernel Github Repo to implement our serial agorithm, then we implement OpenMP and MPI based on the serial version.

### 2.1 OpenMP

We first use OpenMP to parallelize the above algorithms. For the computation of charged particles' positions and velocities, we just parallelize by particles, where each worker grabs a portion of charged particles and performs the computation based on the electronic field. For the update of the electronic field, we adopt two ways of parallelization: parallelizing by grids and parallelizing by particles.

### 2.1.1 Parallelize by Grids

If we break down the for loop in the first algorithm mentioned previously, the code should contain 3 loops iterating through the 3 dimensions to access a very grid point.

```
for x in (0..L) {
    for y in (0..L) {
         for z in (0..L) {
            ......
         }
    }
}
```

Therefore, this provides us with more room for parallelization. Since the grid points are stored in a row-major fashion, the points on the z-axis should be placed together in memory. If our parallelization could utilize this locality, we could achieve a better speedup.

For this algorithm, we try to parallelize by different axes, including the x-axis, y-axis, and z-axis. Since parallelizing by the z-axis will introduce a lot of cache false sharing, we expect the speedup to be the least.

We use OpenMP's `parallel for` to perform the parallelization here, targeting the GHC machine. Each worker will be responsible for calculating the electronic field for some grid points (either partitioned by x, or y, or z).

### 2.1.2 Parallelize by Particles

This parallelization is based on algorithm 2, where we calculate the contribution of the electronic field for each charged particle and then accumulate all updates for each grid point. Similarly, we use OpenMP's `#pragma omp parallel for` to perform the parallelization on particles. However, since the method entails parallelized updates from different workers for one memory address, we need to wrap the update using OpenMP's `#pragma omp atomic` to ensure that the updates are atomic.

## 2.2 MPI

Based on the above algorithm, we have 2 main data structures: 1D particle array and 3D grid array. The former one stores all the particles, including their position, velocity, charge, etc. And the latter one stores all the grid points in the electronic field.

Intuitively, since MPI cannot share memory, we can let each MPI thread to store and take charge of only portion of these data structures to achieve parallelism and save memory, in case the particle number and grid length is too large to store in a single thread.

So, there are 3 MPI parallel algorithms we can explore:

1. **Parallel by Particles, Broadcast Grid**: Each MPI thread only stores part of the 1D particle array, and take the responsibility to update its own particles and update the grid based on its own particles. Note we must broadcast the whole 3D grid array.

2. **Parallel by grid, Broadcast Particles**: Each MPI thread only stores part of the 3D grid array, and take the responsibility to update the particles located in its own grid area and update the its own grid points. Note we must broadcast the whole 1D grid array.

3. **Parallel by Particles and Grid, Broadcast Nothing**: Each MPI thread stores part of the 3D grid array and the corresponding particles located in its own grid area. It takes the responsibility to update the particles located in its own grid area and update the its own grid points. Our goal here is to broadcast nothing.

These 3 methods can be suitable for different settings. In the following sections, we will introduce them one by one in detail.

#### 2.2.1 Parallel by Particles

During the initialization phase, firstly we initialize all the particles, and divide them equally to each MPI thread. Secondly we initialize the local electronic field grid based on each thread's own particles, and reduce each thread's local updates for the grid array.

During the main loop, 1) each thread first updates its own particles based the broadcast grid, 2) each thread updates the local electronic field grid based on its own particles, 3) we reduce the updated local grid array and broadcast it via MPI_Reduce and MPI_Bcast.

Notably, the first 2 steps are computation, and the last step is communication. During the experiments, we find this method work well for the setting of small grid size and large particle size, where the communication time is little due to the small grid array and workload is balanced because of evenly partitioned particles. For example, with 16*16*16 grid size and 200000 particles, the communication cost is only 8%.

However, if we have rather larger grid size, the communication phase becomes the bottleneck, taking up about 40% time when grid size is 256*256*256, and particle size is 200. So, we need to find a better method to handle this situation, which is "Parallel by Grid".

### 2.2.2 Parallel by Grid

During the initialization phase, firstly we initialize all the particles, and broadcast them to each thread. Secondly, we evenly divide the grid cube into several parts, the number of which is equal to the number of processes, and then each thread mallocs and initializes its own grid area based on the broadcast particles.

During the main loop, 1) each thread only updates the particles in their own grid area, otherwise skips it, 2) we gather each thread's updated particles via MPI_Gatherv and then broadcast it, 3) each thread uses the broadcast particles to update the grid points it takes charge of.

The first and last steps are computation, and the second step is communication. This method is suitable for small particle size and large grid size, and it takes only the half of time compared to the first method under 256*256*256 grid size and 200 particles, with 0.1% communication cost. But it performs worse under 16*16*16 grid size and 200000 particles.

### 2.2.3 Parallel by Particles and Grid

Although the first two approaches complement each other to some extent and can cover most situations together, what if both grid size and particle size are large? For the setting with 256*256*256 grid size and 20000 particles, the first 2 methods take too much time. Hence, we want to find a way to parallelize by both particles and grid.

However, to update the electronic field grid, we need all the particles to update each grid point, but it will be infeasible to broadcast all the particles as our goal is to avoid broadcasting all the particles or the whole grids, considering the particle number and grid size may be very large. So we figure out a compromise solution by using the second algorithm stated in the background section. In short, we only use neighbour particles to update the grid.

Although this setting will reduce the accuracy to some degrees, it should be acceptable because the update value is inversely proportional to the distance between the particle and the grid point, and particles far away will not influence current grid point a lot, which can be ignored. The main incentive here is this new setting allows to parallelize by particles and grid together, thus increasing the speed a lot, without compromising too much accuracy.

This method is similar to the "Parallel by Grid" method, but without broadcasting the whole 1D particle array and only uses neighbour particles to update the electronic field at every grid point. In short, each thread only stores and takes charge of its own grid and the particles located in its own grid area, so particles should sent among threads, either its location changes to a new grid area or it is needed for another thread to update its grid.

So during the initialization phase, apart from dividing the grid equally, initializing particles in each thread's own grid area, and initializing each thread's own electronic field, we also need to initialize 2 types of send buffers and 2 types of receive buffers. The first type of buffer pair is for sending and receiving particles that move from one grid area to anther grid area after updating their positions, while the second buffer is for particles that could be used to update the edge of one thread's grid area but are stored in another thread.

Also, for the first type buffer, there are total 26 buffers per thread, since for one cube in 3D dimension, there are 26 cubes "around" it; while for the second type buffer, there are total 8 buffers per thread, since there are only 8 cubes directly "around" one cube. The buffer initialization size is about 1/10 of the local particle size, and can be increased when it is not enough.

During the main loop, 1) each thread first updates its own particles based on its own grid area, and if the particle should belong to anther thread after updating its position, put it into the send buffer of the targeted thread. 2) after updating all the local particles, each thread should send and receive the particles from the first type of buffer, and also each thread should attach the received particles to its own particles. 3) each thread should first decide what particles that needed by the direct neighbour thread to update their own edge grid points, and put these particles into corresponding buffers, then communicate these particles, 4) each thread should use local particles and particles received from step 3 to update the grid points within its own grid area.

The first and last steps are computation, and the middle 2 steps are communication. This method is very powerful. It can handle 5 iterations for 512*512*512 grid size and 20000000 particles within 12.6 seconds, and 1.24 second for 2000000 particles.

## 3 Results of OpenMP

### 3.1 Parallelize by Grids vs Parallelize by Particles

We run the experiment for OpenMP using the cluster initialization method. Specifically, we initialize the charged particles all around one center point, with a radius of 1 (i.e. the charged particles are scattered around 1 unit length around the center point). We run this experiment with a grid size of 16*16*16, and a particle number of 200000, focusing on the setting of a large number of charged particles relative to the grid size. The result is shown as fig. 2.

As we can see, the speedup of parallelizing by grids is slightly better than parallelizing by particles. That should be because of the atomic update of the electronic field in the particle parallelizing method. With the cluster initialization method, it is likely for two workers to update the same grid point, and this will incur overhead on the parallel performance.

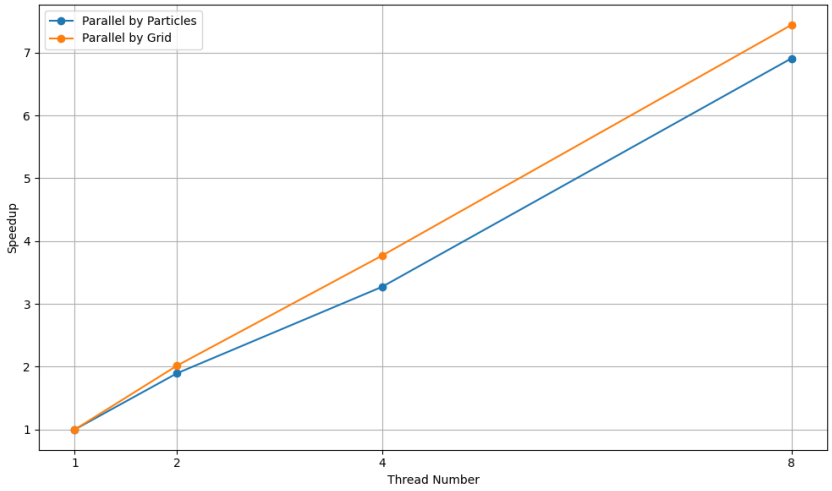


Figure 2: OpenMP Speedup on different axes

### 3.2 Parallelize on Different Axes

As mentioned in section 2.1.1, we have parallelized by grids on the x-axis, y-axis, and z-axis. We run this experiment with a grid size of 16*16*16, and a particle number of 1000. The result is shown as fig. 3.

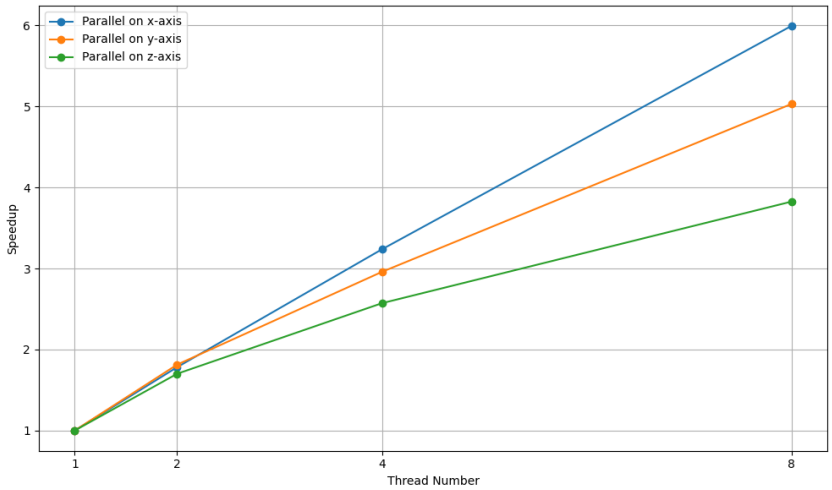


Figure 3: OpenMP Speedup on different axes

As we can see, the speedup is the best when we parallel on the x-axis, reaching 6x. The speedup is the worst when we parallel on the z-axis, with nearly 4x. The reason should be that there is more cache false sharing when we parallelize by the z-axis, considering the row-major storage of the grid points. When we parallelize by the x-axis, the data processed by one worker is the least likely to be present in another worker's cache, and thus the speedup is the best.

# 4 Results of MPI

Due to our 2 settings and 3 versions of MPI methods, in order to control the evaluation workload, we choose GHC machines to evaluate our job. Also for simplicity, we only calculate the speedup based on the main loop, ignoring the initialization time. Unless specified, the iteration number is 2 to save time.

As stated above, we have 2 different settings. The first one is the main setting, which has larger time complexity and more accurate result, while the second one uses less time and has a less accurate result.

## 4.1 The Main Setting

In this setting, when updating the grid, we look at all the particles to decide the electronic field for each grid point. Since the third version doesn't support this setting, we only evaluate the first 2 versions of our MPI algorithms (i.e. parallel by particles and parallel by grid).

### 4.1.1 Speedup Result

**Large Particle Number** There are 200000 particles in total with 16*16*16 grid size.

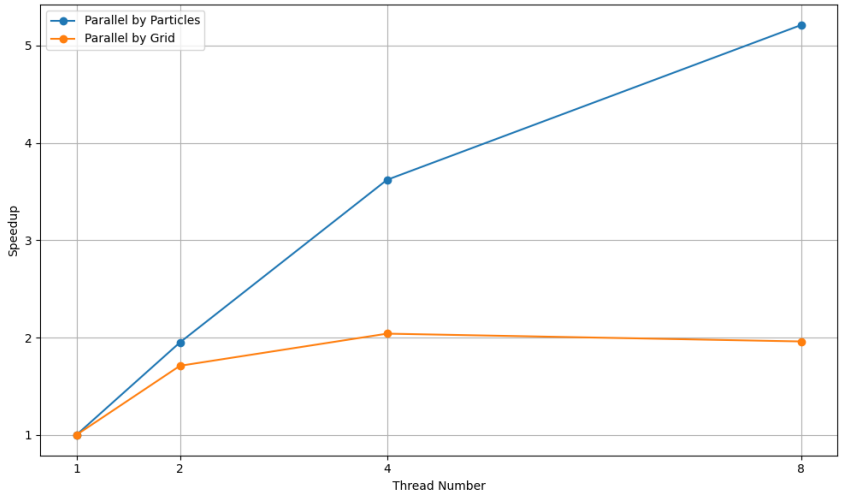


Figure 4: The Main Setting-Speedup Result-Large Particle Number

From fig. 4, we can clearly see the result of "parallel by particles" is better than "parallel by grid". The first reason is that the latter version needs to broadcast and sync the large particles array in every iteration; and the second reason is that during the phase of updating the grid, the time complexity is $O(local\_grid\_size^3 * local\_particle\_size)$, so the latter method spends more time in this phase than the former one.

Also, "parallel by particles" doesn't reach the full speedup, and the reason is because of communication. In each iteration, the update of the whole grid needs to be reduced and broadcasted, and which is unavoidable. As for the workload balance, we also print the total time for each thread, and we find the workload is roughly balanced.

**Large Grid Size** There are 200 particles in total with 256*256*256 grid size.

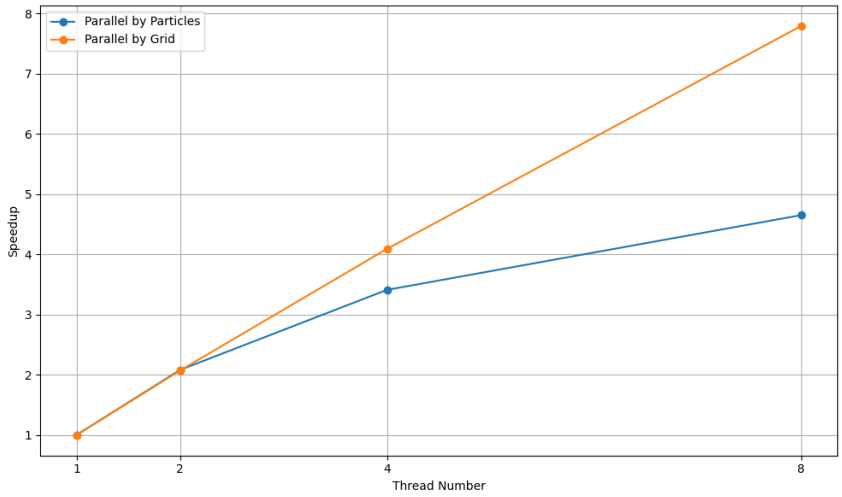


Figure 5: The Main Setting-Speedup Result-Large Grid Size

From fig. 5, we can clearly see the "parallel by grid" method works better than "parallel by particles". The reason is obvious, the latter needs to communicate the update of the whole grid each iteration, and this communication takes up near 40% time under 8 cores.

For the former method, it does achieve the full speedup, indicating this method is very suitable for this scenario.

### 4.1.2 Sensitivity

All the sensitivity experiments are conducted under 8 threads.

**Grid Size** We fix the particle number to 200, and vary the grid size from 100, 150, 200, 250, 300.

From fig. 6, we can clearly see the impact of increasing grid size is definitely larger for "parallel by particles", compared to "parallel by grid", mainly because the skyrocketing communication cost.

**Particle Number** We fix the grid size to 16, and vary the particle number from 100000, 150000, 200000.

From fig. 7, we can clearly see under every large particle number, "parallel by grid" takes more time than "parallel by particles". But the rising slope difference is not obvious of the last experiment, and

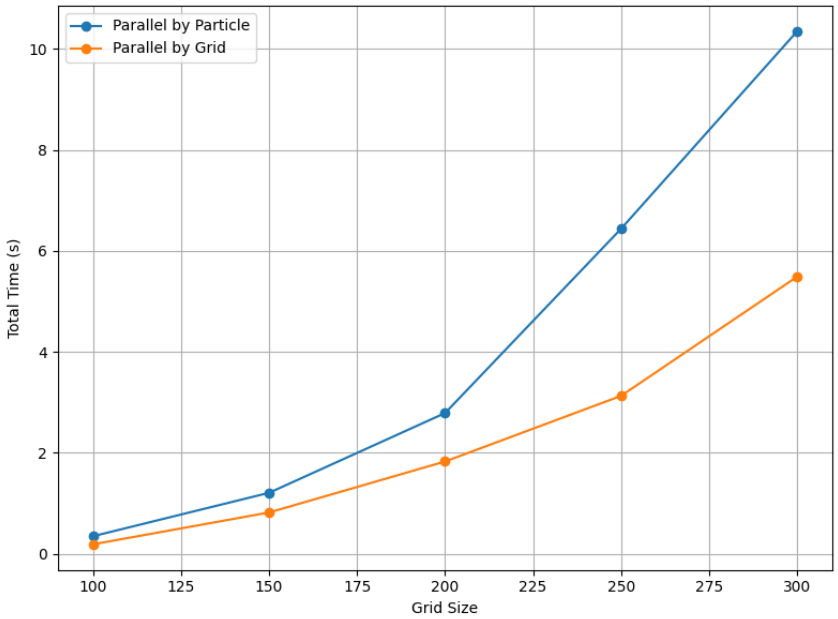


Figure 6: The Main Setting-Sensitivity-Grid Size

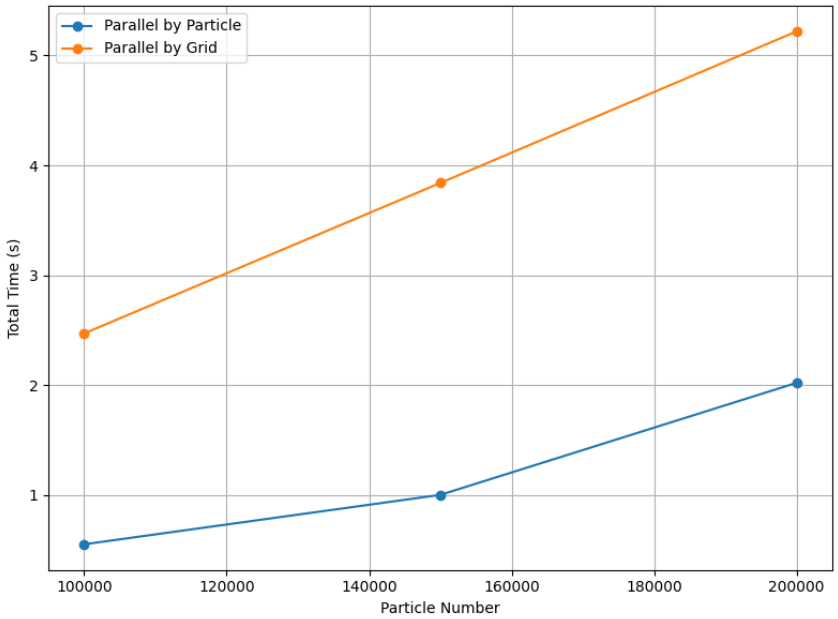


Figure 7: The Main Setting-Sensitivity-Particle Number

it is because the time complexity is $O(grid_size^3 * particle_number)$, so grid size influences the time more dramatically.

**Initial Particle Distribution** All the other results are based on the evenly initialized distribution setting, which means all the particles are scattered equally across the whole grid at the beginning.

We also explore the uneven distribution setting, and find if all the particles are centered around one grid point, with 200000 partcles and 16*16*16 grid size, the "parallel by particles" work much better than "paralle by grid" method, at least for first iteration. In the first iteration, the former method takes only 1.55s, but the latter takes 7.14s. But when there are more iterations, the uneven distributed particles are gradually scattered evenly, making the difference not too obvious.

### 4.2 The Second Setting

We make all the versions using the second setting, and we find in certain situations, the third method is powerful. For example, for 20000000 particles and 512*512*512 grid size, the "parallel by particles" version cannot allocate 512*512*512 memory per thread, and the "parallel by grid" version needs more than 120 seconds to run because of too much communication cost. However, the third version only needs less than 5 seconds to finish.

We only run the speedup experiments due to time limit, and we choose not too extreme configurations to make sure every version can run.

#### 4.2.1 Speedup Result

The experiment setting is 2 iterations, 320*320*320 grid size, and 20000000 particles.

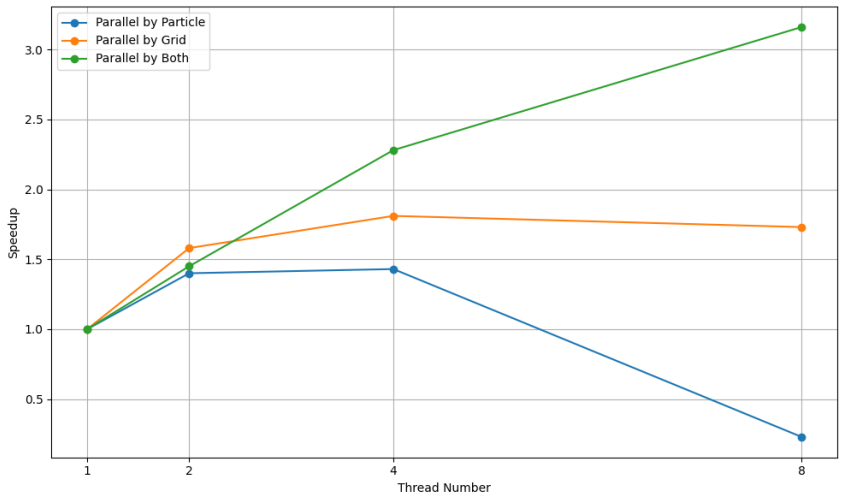


Figure 8: The Second Setting-Speedup Result

From fig. 8, we clearly see that "parallel by both" is better than other 2 methods. "parallel by particles" and "parallel by grid" even cannot achieve speedup when there are 8 threads. The main reason is that the communication cost increases significantly as the thread number goes up, because they have to sync a lot of data among every thread each iteration, as both particle number and grid size are very huge.

For "parallel by both" method, it still doesn't reach great speedup. One reason is that it must communicate points to achieve correctness, so this communication cost is unavoidable. And we also print each thread's time, so the workload is roughly balanced.

Actually the communication time goes down from 2 threads to 8 threads, which is reasonable since when there are only 2 threads, there much more particles to communicate compared to 8 threads. So it is strange that 8 threads don't achieve 2x speedup compared to 4 threads for the computation time. It couldn't be false sharing since it is MPI. And it also seems not related to cache misses because we store 3D array as 1D array in implementation, and always access them sequentially.

### 4.3 Conclusion

Based on the above results, we provide some takeaways for these 3 versions in table 1.

| Version | Handle Large Particle Number? | Handle Large Grid Number? | Handle Uneven Particle Distribution? | Accurate Degree? |
|---|---|---|---|---|
| Parallel by Particles | Yes | No | Yes | Completely Accurate |
| Parallel by Grid | No | Yes | No | Completely Accurate |
| Parallel by Both | Yes | Yes | No | Acceptably Accurate |

Table 1: Comparison of Different MPI Parallelization Approaches

## 5 Work Distribution

Yuanxin Cao: 50%

Lan Lou: 50%

## 6 Project Webpage

The link to the webpage of the project is here.

## 7 References


[1] David Tskhakaya, Konstantin Matyash, Ralf Schneider, and Francesco Taccogna. The particle-in-cell method. *Contributions to Plasma Physics*, 47(8-9):563–594, 2007.

[2] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[3] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[4] J. Derouillat, A. Beck, F. Pérez, T. Vinci, M. Chiaramello, A. Grassi, M. Flé, G. Bouchard, I. Plotnikov, N. Aunai, J. Dargent, C. Riconda, and M. Grech. Smilei : A collaborative, open-source, multi-purpose particle-in-cell code for plasma simulation. *Computer Physics Communications*, 222:351–373, 2018.