Part II. MPI

Xiaoyun Zhao, Nuochen Lyu

1. Overview

MPI utilizes the distributed memory model which allows messages to be exchanged among processors. Our MPI implementation achieved $O(n)$ runtime with $O(n/p)$ scaling, where p is the number of processors.

2. Implementation

2.1. Data Structures

$ class bin_t

We adopted the Bin data structure from the previous serial implementation with additional modifications, the code is shown below (only kept class variables and function signatures):

```
class bin_t{
public:
    std::list<imy_particle_t*> particles;
    std::list<imy_particle_t*> newparticles;

    void add_particles(imy_particle_t * p);

    void splice();

    void clear_newparticles();

    void clear_particles();

    void binning();
    void neighbor_particles(std::vector<imy_particle_t> &res);


    void moved_particles_in_bin(std::vector<bin_t> &bins, int b_it);
};
```

Similar to the serial implementation, each particle in the simulation only checks its neighboring 9 bins (including itself) to interact with, thus ensures a linear runtime. However, binning in MPI is more complicated than that in serial code, since particles in the border area could have interactions with particles in another processors. Therefore, a "neighbor exchange" step is required to correctly compute the forces in MPI. Details will be explained in later sections.

$ class my_particle_t

A simple class defines particle's properties:

```
class my_particle_t {
public:
      double x, y, vx, vy, ax, ay;
};
```

$ class imy_particle_t

This class is basically a wrapper for the original particle type. It takes my_particle_t as a class variable, and adds an index and corresponding bin's information to ease the later particle move and interacting process:

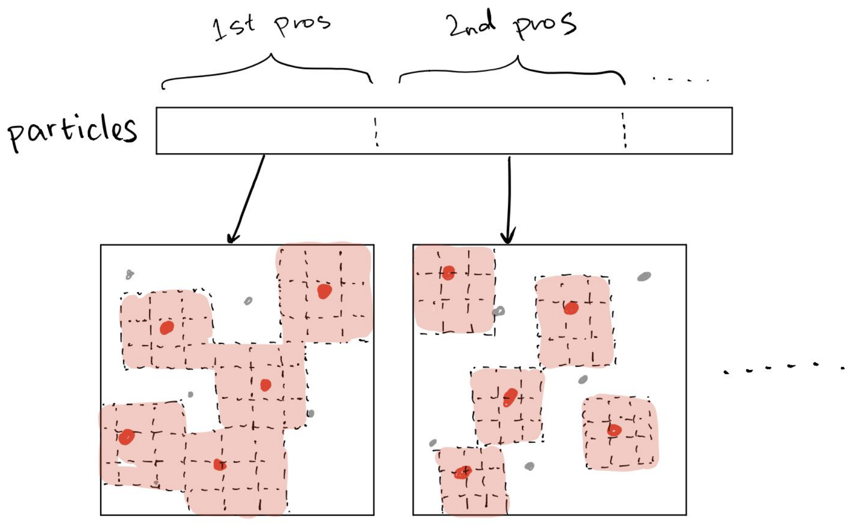
```
class my_particle_index {
public:
    my_particle_t particle;
    int index;
    int bin_idx;
```

```
   void move();

   void apply_force(my_particle_t &neighbor , double *dmin, double *davg, int
*navg);
  };
```

2.2. Initial Attempt

We first aimed to limit the runtime to $O(n)$, so we added the serial code method on top of the naive MPI implementation. Basically, after the MPI gathered all particles from all processors, we applied binning for each particle on the whole canvas, other than checking all particles in the canvas. This attempt was for two reasons: first, to explore how expensive the MPI communication could be, which includes both scattering and gathering processes; and second, to establish a performance benchmark for our later MPI solutions. The performance of this initial implementation was quite decent that we expected. We were able to lower the runtime to $O(n)$. It proved that although expensive, communication overhead is not a determining factor for runtime time complexity compared to the actual computation when n is large.

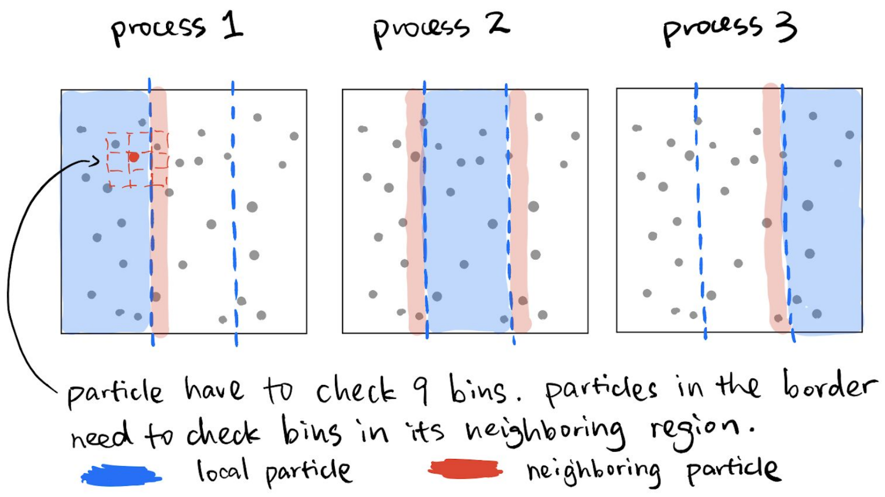


2.3. Major MPI Communications

MPI_Bcast	MPI_Scatterv	MPI_Ibsend	MPI_Recv
used to broadcast two buffers to each processor in the communicator before scattering data	distributed data to each processor	used to exchange border particles with current processor's neighbors in non-blocking buffered send manner	used to receive border particles sent by neighboring processors

The procedures for each time step of the simulation are as follows:

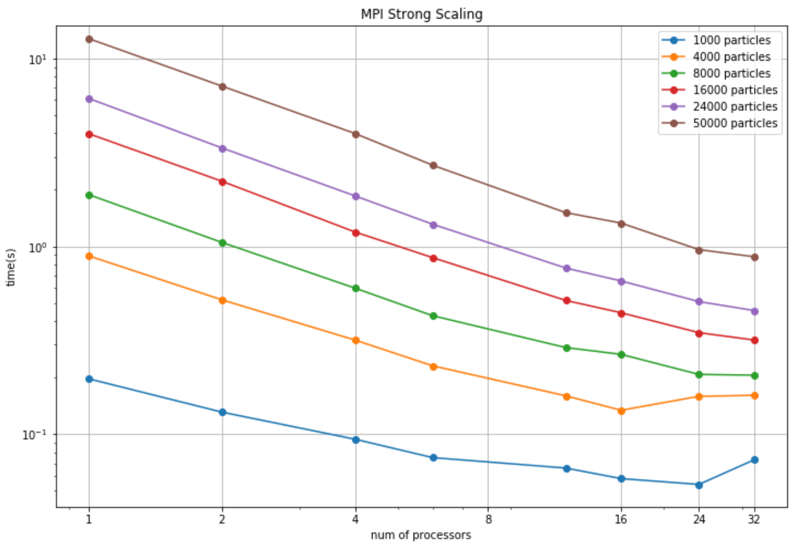
1. Neighboring processors gather border particles and assign to bins
2. Compute forces in each processor
3. Update particles' positions (move particles)
4. Exchange moved particles if they belong to bins in another processors
5. Rebin all particles


process 1
process 2
process 3
particle have to check 9 bins. particles in the border need to check bins in its neighboring region.
local particle
neighboring particle

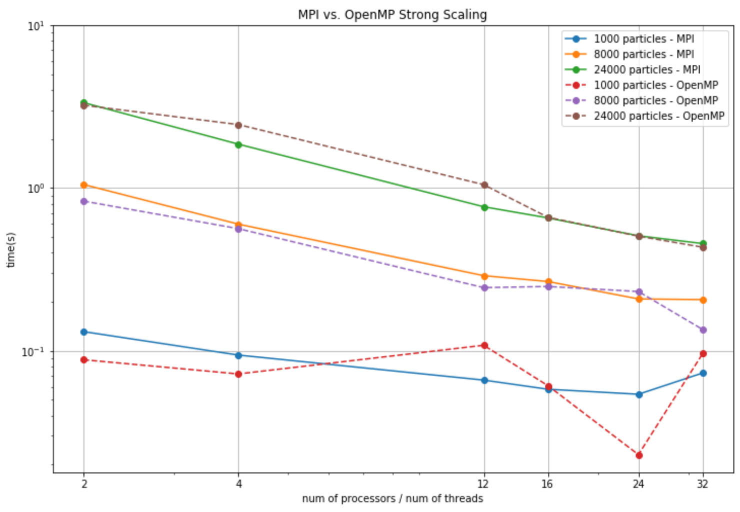
2.4 Results

The figure below shows strong scaling of MPI. We can see clear $O(n)$ runtime and $O(n/p)$ with scaling. We also noticed some outliers that did not follow the linear pattern when n is relatively small (n = 1000, n = 2000) but number of processor is relatively large (p = 24, p =

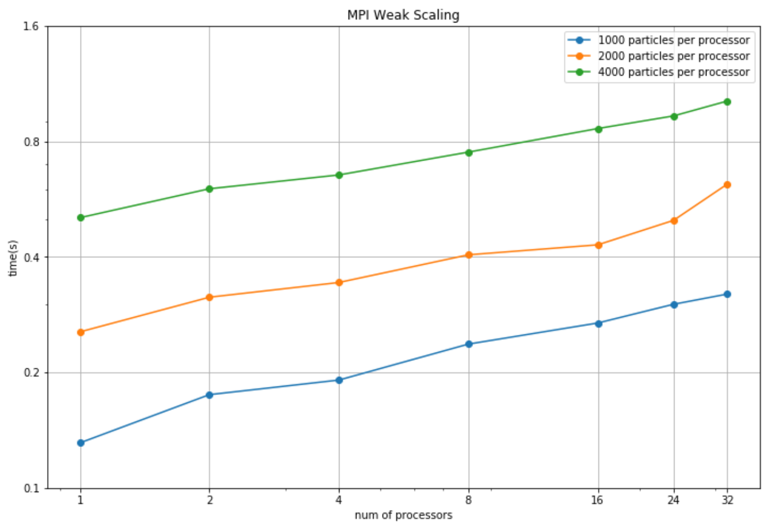
32). Such behavior could be caused by the expense of communication overhead surpassed the actual computation of the simulation at smaller number of particles.



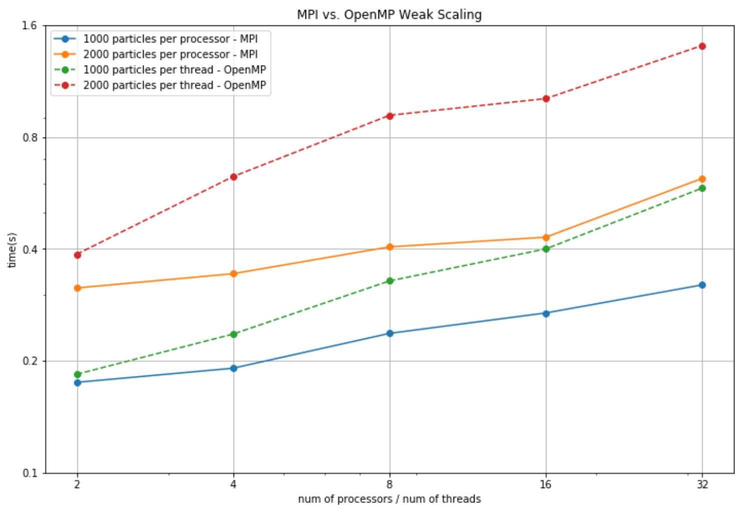
The figure below compares performance between MPI and OpenMP with the same number of processors / number of threads in strong scaling. Both approaches achieved $O(n/p)$ with scaling, and they tend to have similar performance. More discussion on how to choose appropriate solutions will be covered later.



The figure below shows the weak scaling of MPI. It shows $O(n/p)$ time complexity with scaling.



The figure below compares performance between MPI and OpenMP with the same number of processors/number of threads in weak scaling. In this case MPI clearly demonstrates better performance than OpenMP when the number of particles is fixed for each processor/thread.

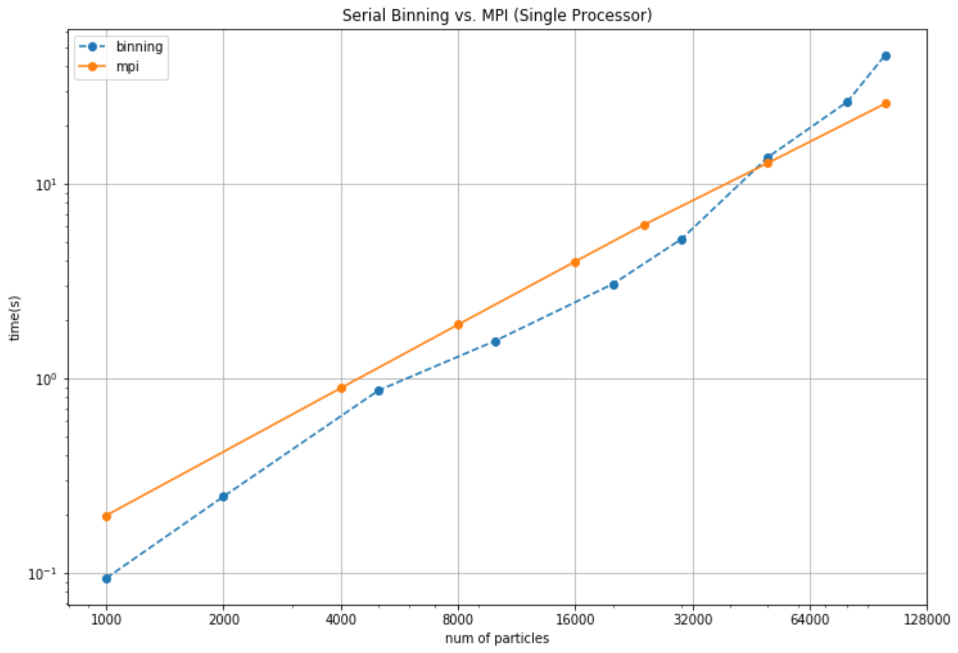


3. Discussion on Using MPI

MPI vs. serial

We first compared MPI with serial binning code. Based on the below figure we can see that, the time that MPI uses to simulate the same amount of particles is slightly higher than the time serial code uses until n = 1,000,000, when MPI obtains a better performance.

We can also observe the fact that MPI displays a clean linear pattern. This quite interesting since MPI has a better linear pattern than the regular single thread code.

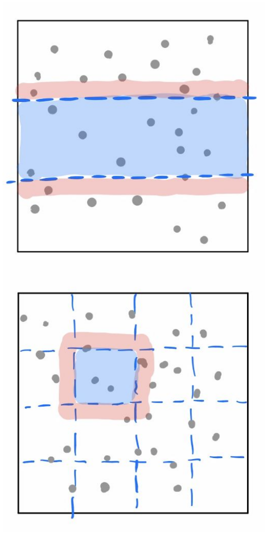


Trade-Offs between OpenMP and MPI

The OpenMP model is based on two concepts: multi-thread and join / fork model of parallelism. Therefore, OpenMP was much easier to implement once the serial code is finished, since they share the sequential part of the execution. This is little needs to modify the code. The only problem is the critical area and racing condition since every thread shares the same memory. Compared to OpenMP, MPI requires a lot of reengineering on the basis of serial code. Basically, each process is like a computer itself. They share the same code but everything else is separated. So communication is a problem. Most of the effort was spent on collecting border particles, interacting with the correct particles in the correct processors, exchanging information, reassigning particles after force being applied, etc. However, the strength of MPI is its scalability. Once the implementation is ready, it can be easily deployed to multiple devices and establish an efficient cluster. Also, MPI could do a better job on computation heavy tasks such as particle simulation by taking advantages of multi-processors.

Besides engineering effort of the implementation, the task itself could also be a determining factor in which parallel approach to choose. Since the communication is expensive in MPI, if the task is computational intensive, which requires great amount of computation resources but limited communication (e.g. training neural networks with big data), MPI could be an appropriate choice. And if the task is I/O intensive, OpenMP could be a more desirable solution.

Shapes for dividing particles



When we distribute the particles to each processor. We decided to slices the canvas into strips other than squares. The main reason is to decrease the MPI communication as less as possible. A square would require 4X times of exchange with neighbors while a bar would require only 2X times of communication.