CSC367H1: Parallel Programming **Instructor: Maryam Mehri Dehnavi**

Project - Part 2 MPI Implementation

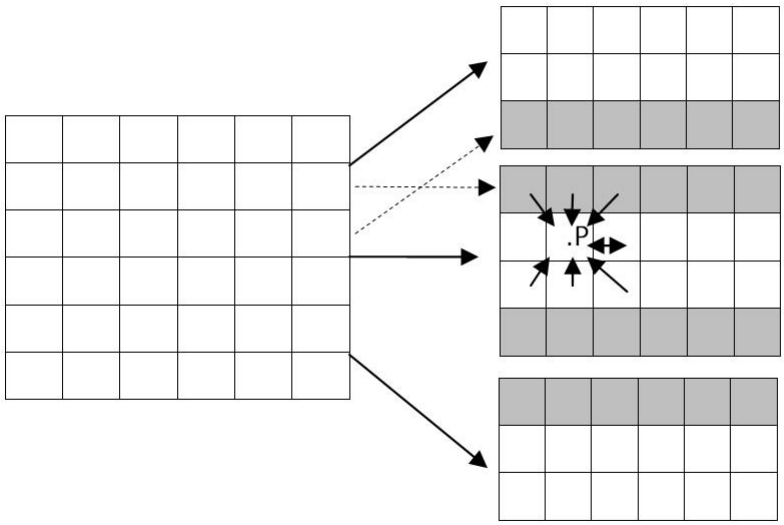
Name: Yifei Tang(1003815437), Zezhong Pan(1002057748)

1 Introduction

In Part 2 of this project, we implemented the parallel particle simulation with MPI, and measure the speedup and scalability.

2 MPI Implementations

Similar as previous phase, we use a data structure that divides the space into a grid, and each small square (with size cutoff $\times$ cutoff) keeps track of the particles lying inside. Each process will be assigned with roughly equal number of consecutive rows of the whole grid and particles in those rows. A process is also assigned with additional row(s) that overlaps with neighboring process(es) as padding. The reason for having this padding rows will be explained bellow. Every iteration of the simulation includes three main steps.



1. building local data structure

Given all the local particles, each process builds its own grid of buckets, and put the local particles into the corresponding buckets. The local gird include two parts, the padding zone and computation zone.

2. computing interactions and moving particles

Each process needs to compute the interactions for the particles in the middle computation zone. This means that particles close to the boundaries, will have to consider points held by other process, which is why the padding zone is needed. The padding zone contains points from the row that is next to the lower/upper boundary, and values of these particles are only read to compute force to the particles in computation zone but never updated. After calculating the interactions, the process move the particles in the computation zone.

3. transfer cross-boundary particles

After being moved, some particles might go beyond the computation boundary of the current process. These particles will be sent to another process. Each process has 4 buffers for sending/receiving particles upwards/-downwards, in order to transfer particles with neighboring processes. After the transfer, a process will update its local particles and get ready for another iteration.

3 MPI Implementation Speedup, Scalability and Efficiency

Setup

Implementation: MPI
Number of processors: 1, 2, 4, 8, 16
Number of particles: 10000 (for strong scaling), 20000, 40000, 80000, 160000

Result

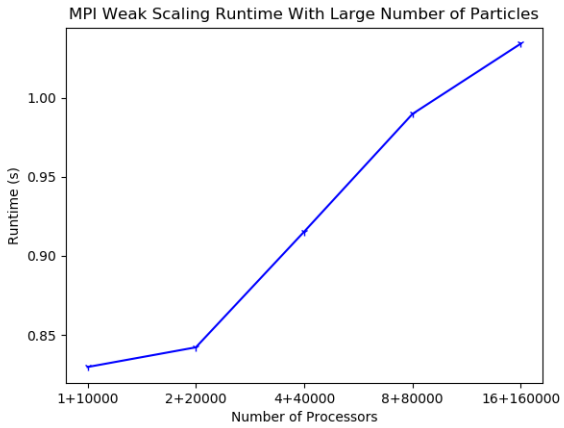


Figure 1: Weak Scaling Runtime

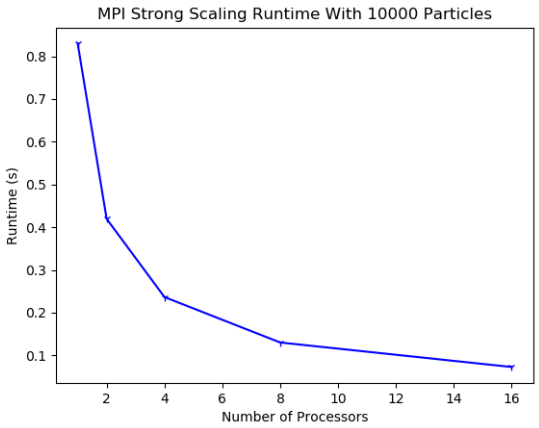


Figure 2: Strong Scaling Runtime

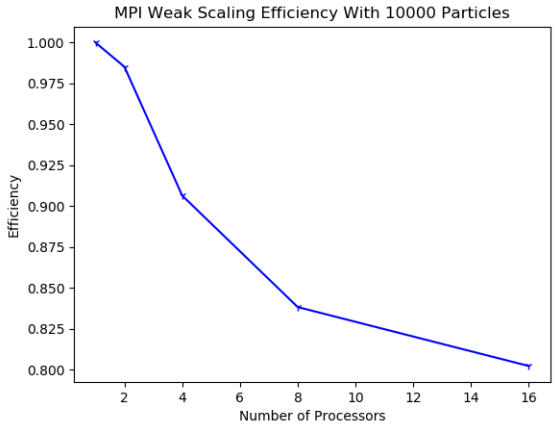


Figure 3: Weak Scaling Efficiency

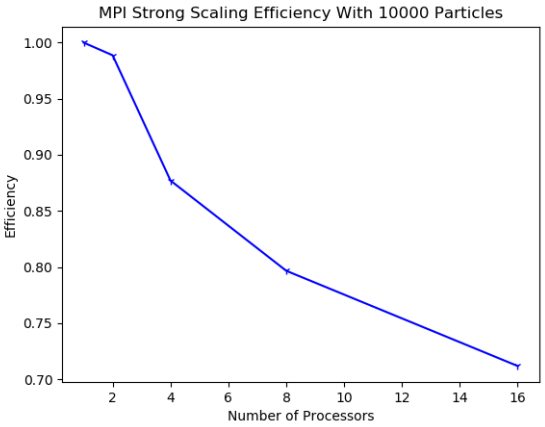


Figure 4: Strong Scaling Efficiency

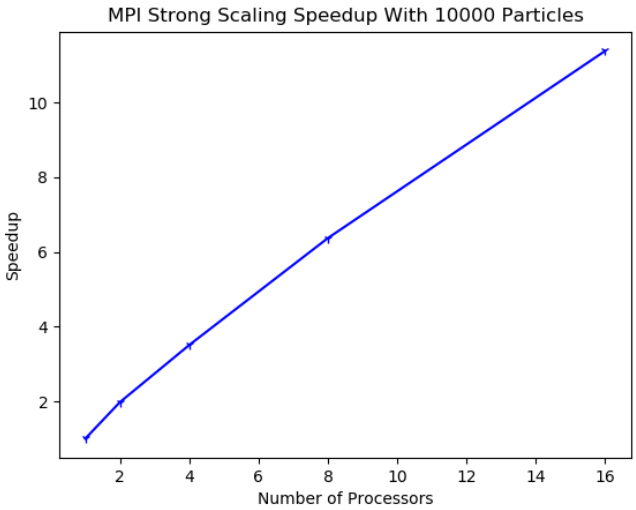


Figure 5: Strong Scaling Speedup

Strong Scaling	Weak Scaling
0.87484	0.90642

Table 1: Average Efficiency

Analysis

The speedup and scalability are very impressive compared with the OpenMP implementation. With the MPI implementation, all the values are stored and computed locally, with no contention dealing with shared data. This is the main reason why MPI implementation is more efficient than OpenMP. The MPI implementation is also more paralleled. In our OpenMP implementation, each thread always has to go over the whole set of particles to build the local data structure no matter how many threads there are, while in the MPI implementation, we improved this part by making each process only keep track of a local set of particles. With this, we achieve a much better strong and weak scalability.

There are still some overhead keeping this implementation from getting to the idealized p-times speedup. One important factor is the communication between processes. In the end of each iteration, the processes has to transfer the cross-boundary particles, and this synchronization is not avoidable. Our early implementation use a naive synchronization method by gathering all the local particles and distribute them again, and this method is much slower due to the larger amount of communication needed and bad scalability. So we minimize the communication by making each process talk only to its neighbors and only transfer the cross-boundary particles. The expected number of particles that cross the boundary each time is relatively low and roughly the same for each boundary. And by using the non-blocking communication method MPI_Isend(), this communication overhead doesn't really increase with the number of processes, and is rather low.