CS267 Homework 2 (Part 2)

Parallelizing a Particle Simulation

Team members: Tzu-Chuan Lin, Chin-An Chen, Byron Hsu

Abstract

The task is mostly similar to homework 2-1; however, for the parallel speedup, we are allowed to use only distributed memory parallelism, which is MPI. That is to say, we only use multi-processing rather than multi-threading on shared memory.

Basically, we use the same algorithm with homework 2-1, and we use MPI to make the algorithm applicable to multi-processing scenarios. In brief, We leverage MPI_Send and MPI_Recv for point-to-point communication, MPI_Gather and MPI_Alltoall for collective communication, and MPI_barrier for synchronizations. In the end, we achieve 10.8396 sec under 1.5 million particles running on two nodes using 68 cores per node as our best result.

Implementation

Overview

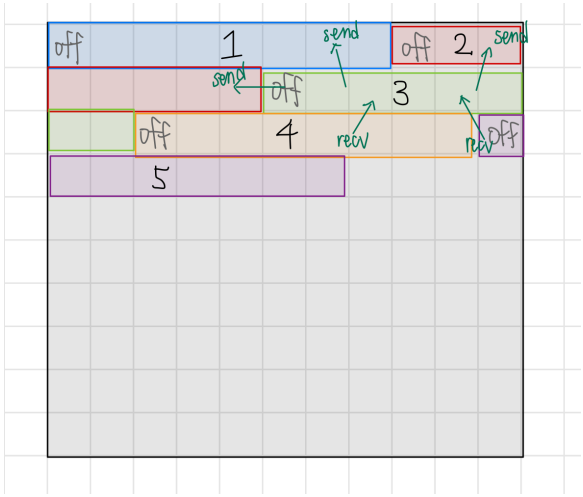


Figure: Overview of our data structure

The grey squares on the figure are bins, and the color rectangles are the bins owned by specific processors. For example, there are 5 processors shown on the figure, and each processor

manages 8 bins. There is an important property “offset” (marked as “off” on the figure) in each processor, which is useful for locating the starting index of the bin owned by my processor.

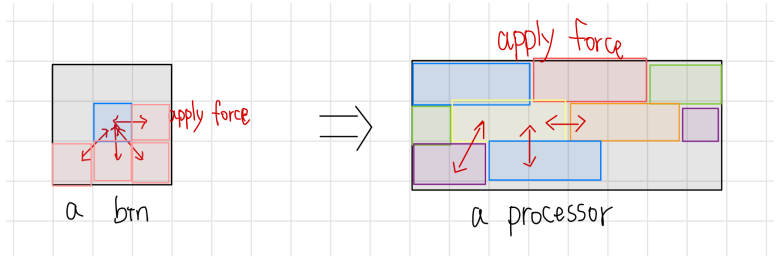


Figure: From serial bidirectional method to MPI bidirectional method: Bins only interact with the bottom and right ones because we apply force on two sides once we load two particles

The arrows with “send” and “recv” on the first figure also matter a lot. For each bin, we only apply the force between itself and its right and bottom ones because we do it bi-directionally, so the left and top ones would be already calculated as the forces for the current bin.

Therefore, for each bin, it only needs to receive the bin information from the right and bottom but does not need the information from other sides. Because the top and left bins also want to receive the bin information from their right and bottom bins, and these bins include the current one, we need to send the current bin to the left and top bins.

Extending to multiple bins scenario in one processor, we want to send the (x,y) information of the current bin to the left and top bins and receive the (x,y) information from the right and bottom bins.

Global variable

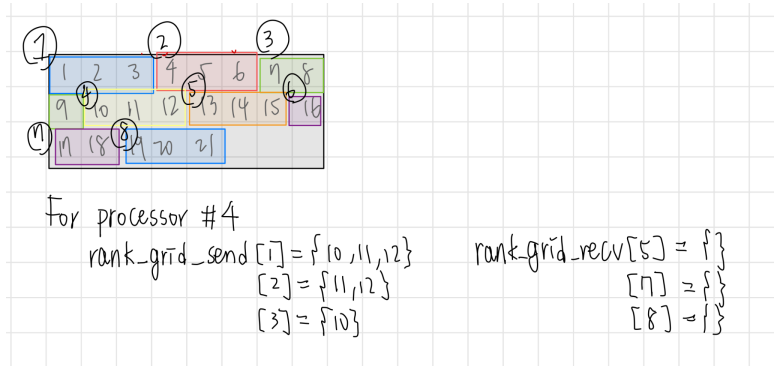


Figure: visualization of rank_grid_send and rank_grid_recv

There are three important global variables: `rank_grid_send`, `rank_grid_recv` and `local_bins`. The data structure of them is `std::map<int, std::vector<int>>`.

For rank_grid_send, the key is the processor rank that is above or on the left of my processor, and the value is a vector containing the indices of bins(or bin) that are above or on the left of my bins owned by that processor.

For rank_grid_recv, the key is the processor rank that is below or on the right of my processor and the value is a vector containing the indices of bins that are below or on the right of my bins owned by that processor grid it will receive.

For local_bins, these are the bins we managed inside a processor, including the current bins and neighboring bins (we will also store the bins owned by our neighbor processors. We will compute the forces for these particles in the bidirectional case but we will not perform *move()* function for the particles inside these neighbor bins because each processor should be responsible for moving the particles they own).

Init Simulation

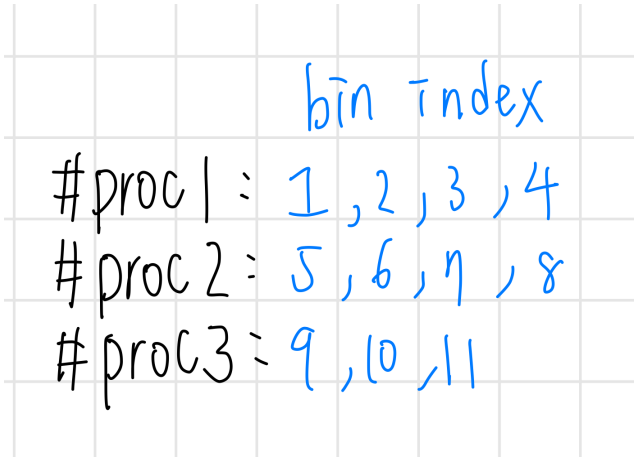


Figure: visualize work distribution over processes

The first step is to compute how many bins each proc will own. If there are `n_bin` bins and `n_proc` processors, we distribute the bins evenly to processors. If `n_bin` is not divisible by `n_proc`, the first `n_bin % n_proc` process will own one more bin than other processes.

In the second step, we compute the bins we want to send and where to send them. Also, we compute where to receive the bins. To elaborate, we will send the bin information to the processor on the left and top, and receive the bin information from the right and bottom.

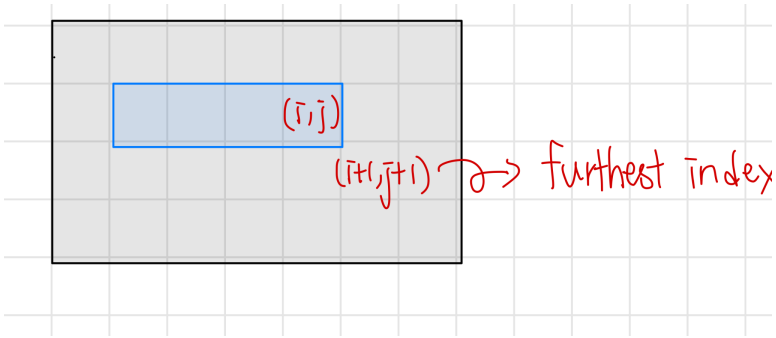


Figure: furthest index

At the third step, we store the bins that will be used in `local_bins`. There are two types of bins in a processor, current bins and neighboring bins. For the current bins, we are responsible for updating the (x, y) and (ax, ay). For the neighbor bins, we only need to update (ax, ay).

How can we categorize the current and neighbor bins? The bins with index [local offset, local offset + num current bins - 1] are current bins, and the bins with index [local offset + num current bins, local offset + num current bins + num neighbor bins] are neighboring bins.

The formula of calculating `num neighbor bins` is the furthest index - local offset - num current bins

Simulation One Step

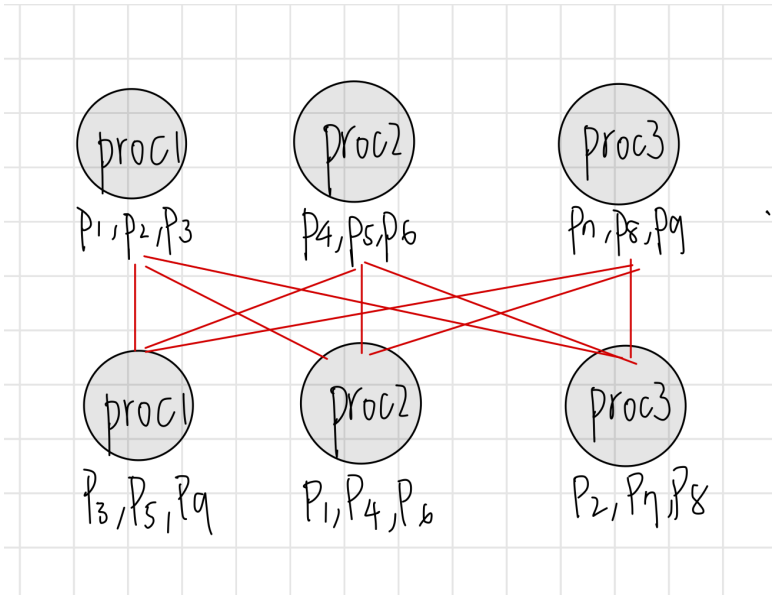


Figure: MPI Alltoall

At the first step, we send bins to top and left (in `rank_grid_send`) and receive bins from the bottom and right (in `rank_grid_recv`).

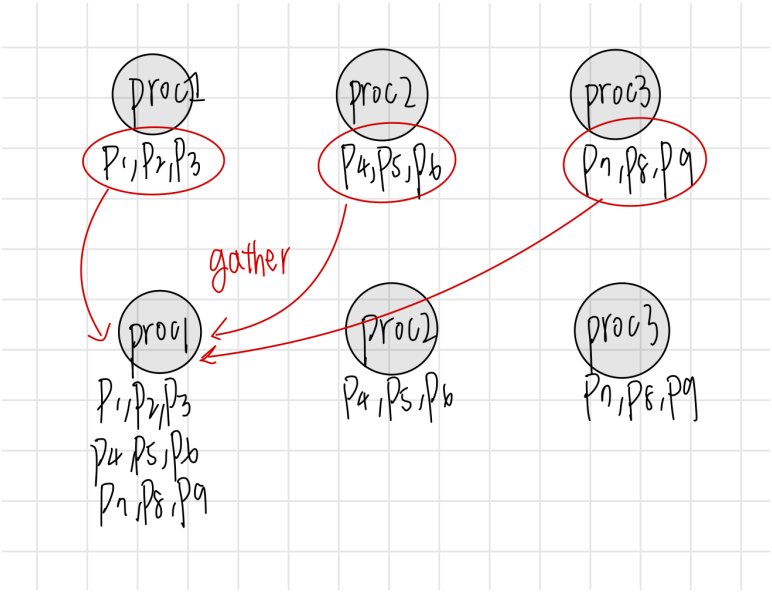
In the second step, we apply force on current and neighboring bins in this process. (ax, ay) will be updated in these bins.

At the third step, we send (ax, ay) to the right and bottom bins (send to the processor that is adjacent to my processor).

In the end, we move particles in current bins. There are three cases:

1. In the same bin: do nothing
2. Move to different bins but still in the same processor: move to other bins locally
3. Move to a different bin that locates in a different processor: use `MPI_Alltoallv` to send particles into new bins

Gather for Save

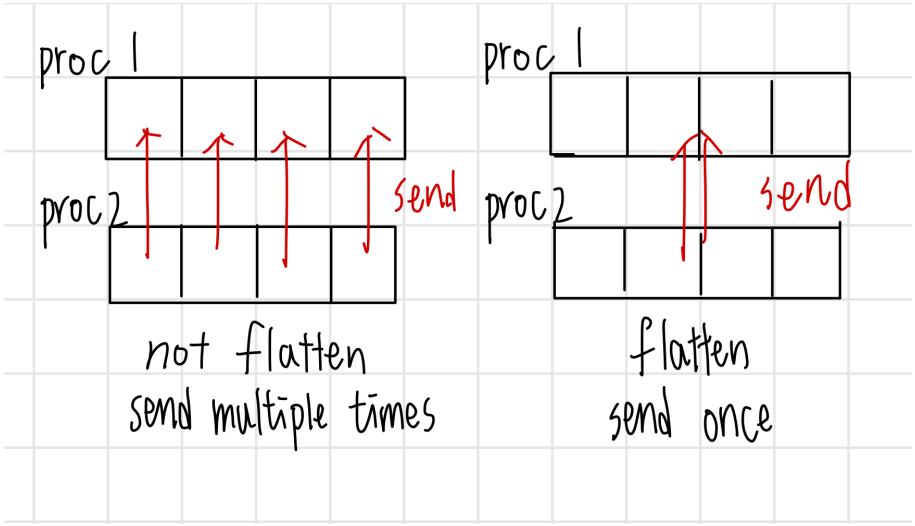


At the first step, we gather particles together from all processors by `MPI_gather`.
Later, we update new particles in `*parts`.

Experiments

- Method 1: Bidirectional:
    - As same as the description above

- Method 2: No Bidirectional:
    - Collecting the particles from every surrounding bin, and updating my own bin only.
- Method 3: Bidirectional & flattening:

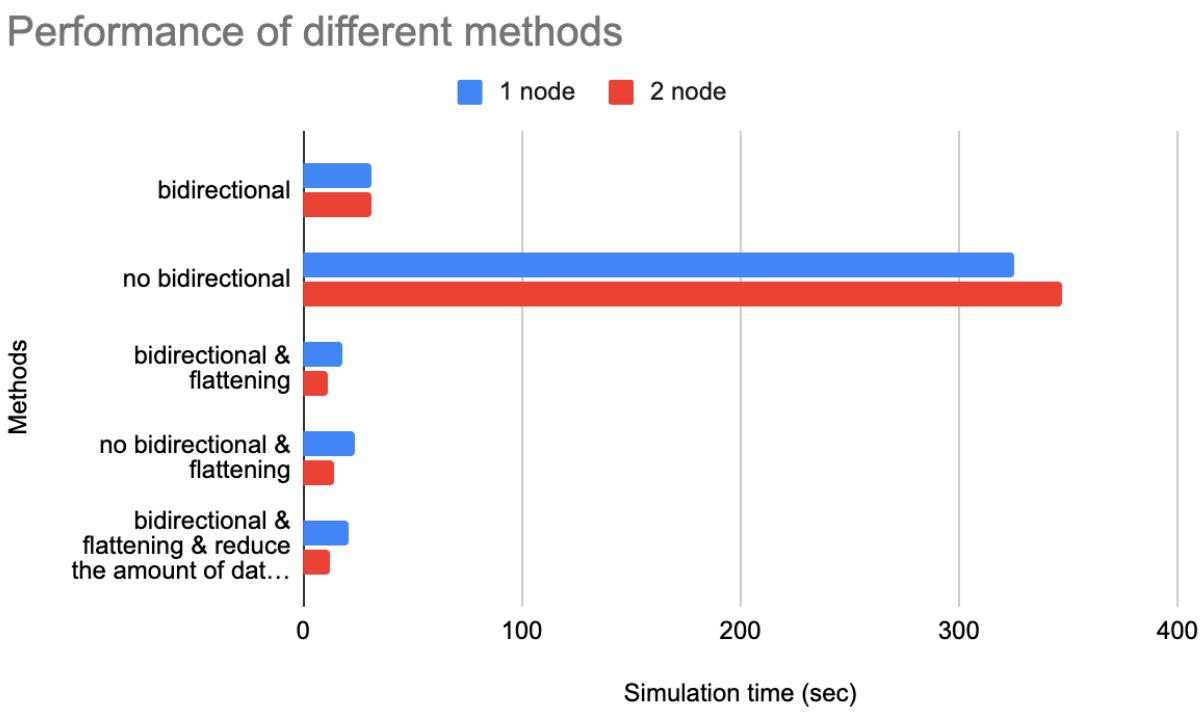


    - Compared to Method 1, to reduce the total communications, we collect all the bins and particles that want to be sent to another processor first and then send to the destined processor only once.
- Method 4: No Bidirectional & flattening:
    - Collecting the particles from the surrounding bins, and only calculating the acceleration for the particles in the current bin. For the communication between processors, we use the flattening method as described above.
- Method 5: Bidirectional & flattening & reduce the amount of data in message-passing
    - On top of the bidirectional and flattening methods above mentioned, we tried to cut down the size of the messages to be sent by only sending and receiving the coordinates and acceleration instead of sending and receiving the whole particles.
    - However, the result turned out to be slower than method 3. That might be because, in order to change the size of the message, we sometimes have to use if-else to switch the data structures we want to use before each send, which would cause the overhead and possibly reduce the cache efficiency (because possibly different data structures might be far way from each other).

***srun -N {1,2} –ntasks-per-node=68 ./mpi -n 1500000 -s 1*
Performances (seconds)

method	1	2	3	4	5
1 node	30.8809	325.335	**17.5079**	23.8832	20.4212
2 node	31.646	346.906	**10.8396**	14.195	12.2506

Method 3: Bidirectional with flattening was the fastest algorithm among the five algorithms



Speedup plots for MPI code approach (strong and weak scaling) *2nodes and 1,500,000 particles

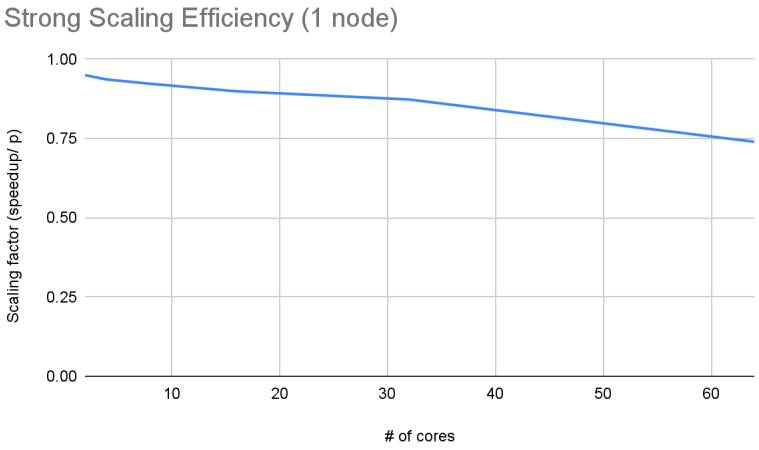
Strong Scaling
The time of the fastest serial code: 888.876 seconds

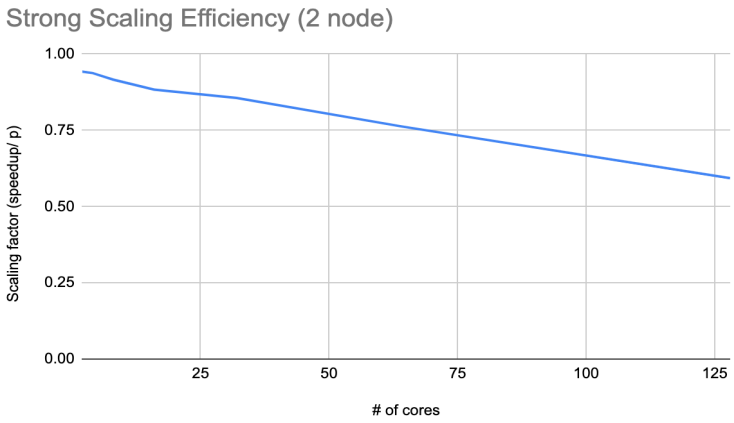
1 node: *srun -N 1 --ntasks-per-node={# of cores} ./mpi -n 1500000 -s 1*

# of cores	2	4	8	16	32	64
Time (sec)	467.875	237.335	120.393	61.7913	31.8104	18.7749
speedup	1.9	3.7	7.4	14.4	27.9	47.3
Scaling	0.95	0.94	0.92	0.90	0.87	0.74

2 node: *srun -N 2 --ntasks-per-node={# of cores} ./mpi -n 1500000 -s 1*

# of cores per node	1	2	4	8	16	32	64
# of cores	2	4	8	16	32	64	128
Time (sec)	471.427	236.978	121.253	62.8894	32.4414	18.201	11.7071
speedup	1.9	3.8	7.3	14.1	27.4	48.8	75.9
Scaling	0.94	0.94	0.92	0.88	0.86	0.76	0.59





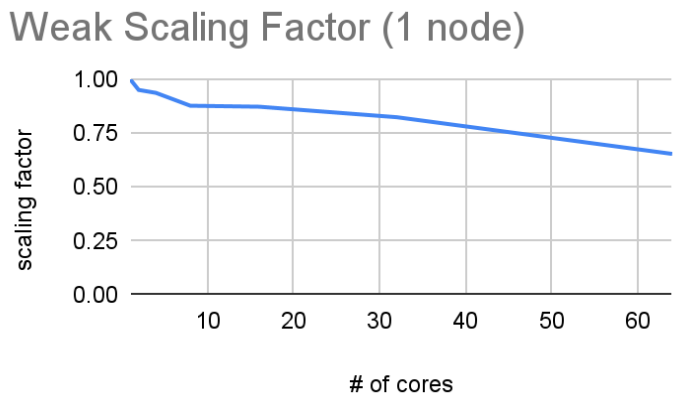
Weak Scaling

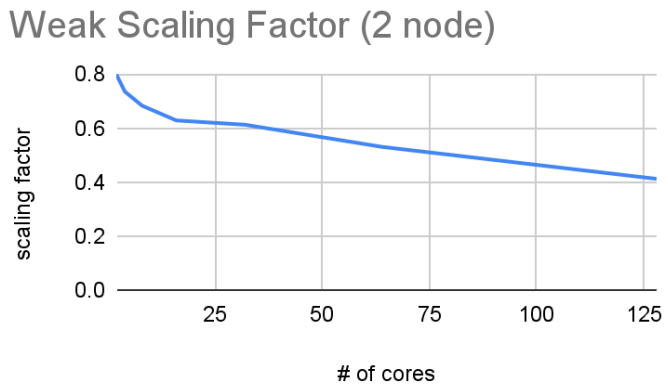
1 node:

# of cores	1	2	4	8	16	32	64
# of particles	10000	20000	40000	80000	160000	320000	640000
Time (sec)	5.6	5.9	6.0	6.4	6.4	6.8	8.6
scaling	1	0.95	0.94	0.88	0.87	0.82	0.65

2 node:

# of cores per node	1	2	4	8	16	32	64
# of cores	2	4	8	16	32	64	128
# of particles	10000	20000	40000	80000	160000	320000	640000
Time (sec)	2.8739	3.12272	3.36026	3.65279	3.74748	4.32758	5.57032
scaling	0.8	0.74	0.68	0.63	0.61	0.53	0.41





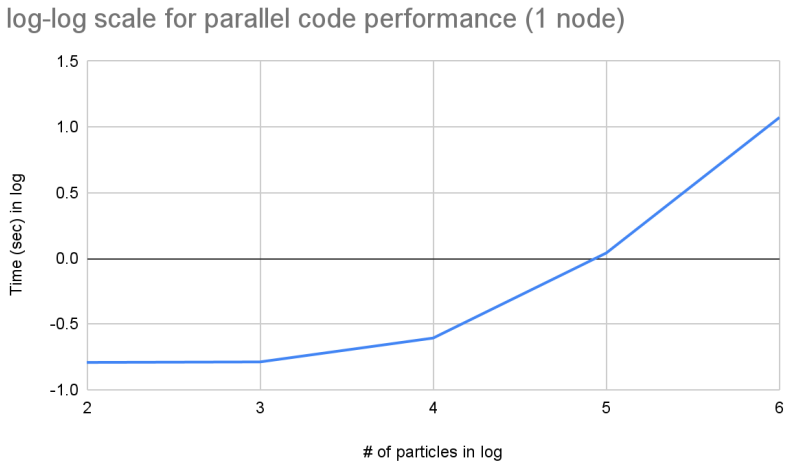
log-log scale for parallel codes performance

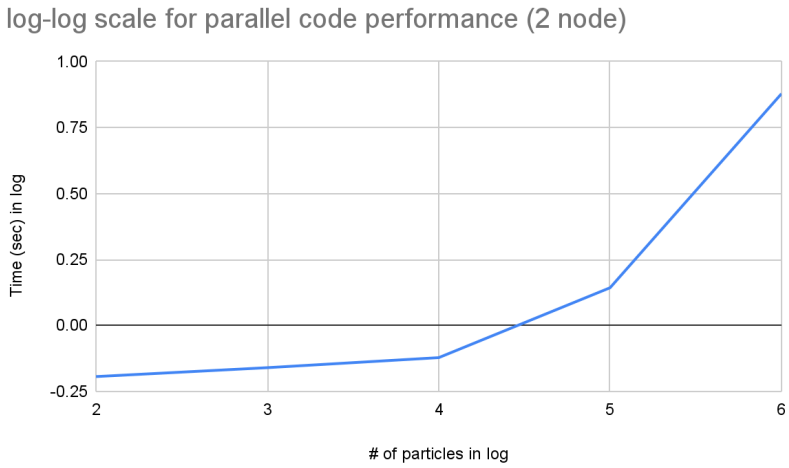
1 node: *srun -N 1 --ntasks-per-node=68 ./mpi -n {# of particles} -s 1*

# of particles	100	1000	10000	100000	1000000
Time (sec)	0.161625	0.163095	0.247727	1.10004	11.8257

2 node: *srun -N 2 --ntasks-per-node=68 ./mpi -n {# of particles} -s 1*

# of particles	100	1000	10000	100000	1000000
Time (sec)	0.640243	0.692009	0.755319	1.39015	7.5557





Where does the time go? How do they scale with p?

Why does the time not perfectly scale with the number of processors? It is because there is a communication and synchronization overhead during the process.

Firstly, we use sending and receiving to transmit information between processors. In each simulation step, we need to spend time on sending and receiving. Also, we will wait for send and receive to finish to move on to the next step.

Secondly, when we distribute the particles into new bins, we need to call Alltoall to send the particles into different bins. By doing so, we need to spend time on sending and receiving particles between processors. Also, there will be an implicit barrier to waiting for finishing.

Lastly, when we gather the particles, we still need to spend time sending and receiving particles. Furthermore, the program will be blocked until all the processes finish Gather function call.

Contribution among the team

Jim: Code implementation
Byron: Implementation and communication in the report
Daniel: Other parts in the report