CSC367 Project Part 2 Report

Farzaneh Tabandeh   Sitao Wang

March 2021

1 Distributed Method to Particle Simulation in MPI

1.1 Partition of Data

In the particle simulation problem, the particles only interacts with the particles that are close to it within a distance of cutoff, thus we choose to partition the particles with respect to their positions in space. The particles are divided into rectangular blocks(or chunks) with equal sizes, and the number of rectangles(or chunks) equals to the number of processors. Thus, each processor owns the block, and will compute the force and new positions for all particles in the block. However, the particles near the boundary of each block also interact with the particles in the nearby block, so each processor also stores the particles in the neighboring blocks that is within the cutoff from the boundary. We call these boundary particles the halo/ghost elements. Notice each processor does not compute the force applied the halo particles, this is the job of the processor that owns those particles specifically.

1.2 Computation withing Partition

For each iteration, each processor computes the movement of particles that it owns. To further leverage the fact that every particle only interacts with particles within cutoff, we use the same algorithm in our sequential implementation in part 1 of the project. For each processor, we divide its block and the nearby space within cutoff into grid cells, and each grid cell has the side length of the cutoff. Particles in a grid cell only interacts with the particles in its own grid cell and the neighboring eight cells. The grid cell array is built for each processor at the start of the program, and in each iterations, the forces are computed one each particle within a cell and moves either within the current processor's grid array or is communicated to the goal process.

1.3 Communication of Moved Particles

As described in previous step, particles are moved after each iteration. However, some particles moved outside of the block and are now owned by other processor.

They could also move close to the boundary and so are now the halo element of another processes in the neighbour block. Thus, after each iteration, processors need to communicate to each other about the particles that are now needed by other processors.

For each iteration, we send the particles moved outside not only to its owner, but also to the neighbor processes if it is in their halo zone. So we decided to send out these two types of particles by one time of communication. Otherwise, these data would need to be sent in two separate communications, adding more overhead in transferring the data. So we send the particles that are now owned by other processes along with all particles that happen to locate in their halo zone in one go.

The challenge is that the number of particles that are communicated are dynamic for each iteration. The `MPI_Alltoallv` provides the functionality of communication of array with different sizes, but it requires the receiver to also know the sizes to be received. Thus, we need to first initiate a call to `MPI_Alltoall` to exchange the size of particle arrays to be transferred, and then use these information to call the `MPI_Alltoallv` to actually send out the particles to each process.

1.4 Overall Algorithm

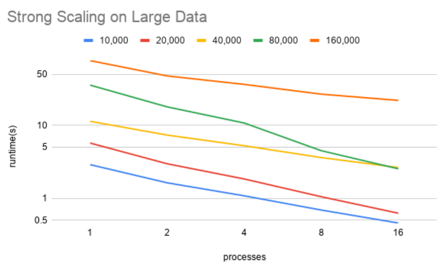
Our algorithm are summarized as follow.

1. Partition the particles into number blocks one per process
2. Segment the block into a grid of cells
3. Compute the forces on owned particles by the process
4. Move the particles to the new grid cells and then check if they still belong to the current process or need to be communicated
5. Communicate/send out the moved particles along with the particles that are considered the halo elements of the neighbor processes
6. Add the received particles into the grid
7. Go back to step 3

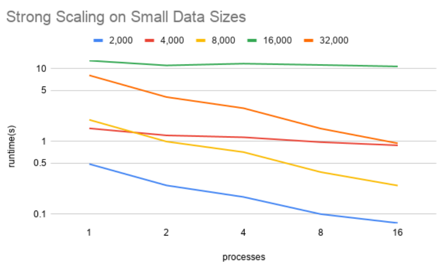
2 Experiments and Results

2.1 Strong Scaling

As Figures 1 demonstrates, despite the communication based parallelization required in this problem, the type of the problem still has a good scaling potential. We can see the overall runtime decreases when we scale the processes from 1 to 16, in both large and small data sizes. Although this simulation problem requires a great amount of communication in each time iteration, we can see that

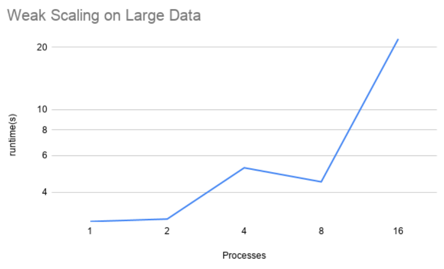


(a) Large Particle Sizes

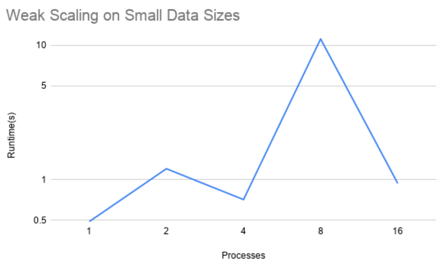


(b) Small Particle Sizes

Figure 1: Strong Scaling



(a) Large Particle Sizes



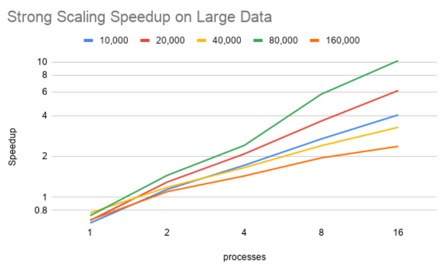
(b) Small Particle Sizes

Figure 2: Weak Scaling

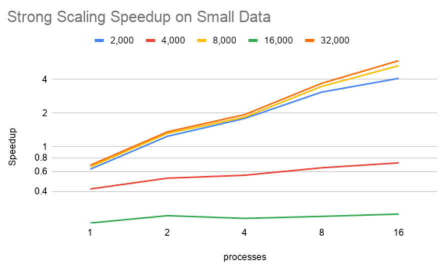
the computation aspect of the problem is so great that scaling to more number of processes could help the performance if number of processes chosen wisely. Our MPI implementation achieves the runtime of around 21 sec for 160,000 data size, although we believe we could have obtained more performance gain if we got to use the non-blocking communication. One observation is that the decrease of runtime in small data size of 4000 and 16000 has quite less steeper slope compared to all other data sizes. Also the overall runtime is these 2 cases is higher than their counterpart big sizes. For instance 16000 data size has much bigger runtime that 32000 data size. The reason is that in these cases the amount of computation needed to be done on the particles is not that much with respect to the amount of communication needed. Note we gain performance if the amount of computation saved by sharing the work load is much more than the overhead of the communication between the processes.

2.2 Weak Scaling

When the problem that we are dealing with is heavily or rather completely computation bound, we expect to see a rather constant runtime when we do weak scaling. That is because with increase in data size, we have also increased



(a) Large Particle Sizes



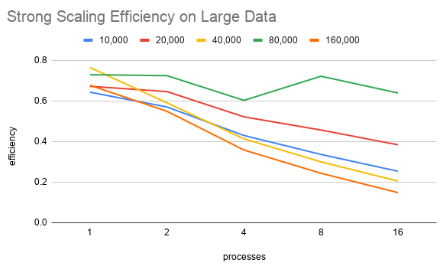
(b) Small Particle Sizes

Figure 3: Strong Scaling Speedup

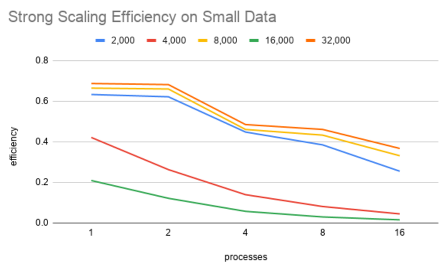
the processes needed for computation on the data. However, since our simulation problem is not completely computation bound, we see while we do weak scaling, the overall runtime still goes up. The increase in runtime is because with increase in number of processes, the overhead of communication between them affects the time complexity. This is shown is Figure 2. Two exceptions in overall trend can be observed in both large and small data sizes. In large data sizes, when we increase the number of processes from 4 to 8 while also increasing data sizes, the runtime decreases greatly. This also happens in small data sizes when we increase processes from 2 to 4. Again this is because in these cases, the number of computation needed compared to the cost of communication of data is much more greater. In other words, the amount of computation that is shared to be done by each process dominates the incurred cost of data transfer and so we get a better runtime and we get closer to the ideal runtime benefit expected by a weak scaling.

2.3 Strong Scaling Speedup

As also discussed in runtime performance, we can see in Figure 3 that the problem is overall scalable and can benefit from increase in number of processes to share the work. We can see overall that the speedup value is at max around 62% of the ideal p value(where p is the number of processes used). It is certainly hard to obtain the speedup value of p, as the problem is not computation bound only. However, for some data sizes specifically we can observe a much better speedup while some sizes does not show a good performance gain over the increase of processes. For instance, a data size of 80,000 provide us with a speedup of around 10 on 16 processes(i.e, it reaches its 62% of the ideal speedup of p = 16) and speedup of around 5.7 on 8 processes(i.e, it reaches its 62% of the ideal speedup of p = 8). This is while on small data sizes the effect of increase in number of processes is not much evident. Specifically in case of data size of 16,000 the speedup remains around 0.2 irrespective of the number of processes used. This very huge differences is because of the way size of the data influences the computation load of the problem. In larger data sizes, more

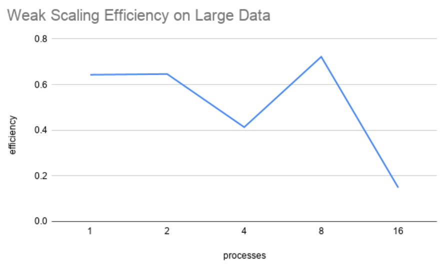


(a) Large Particle Sizes

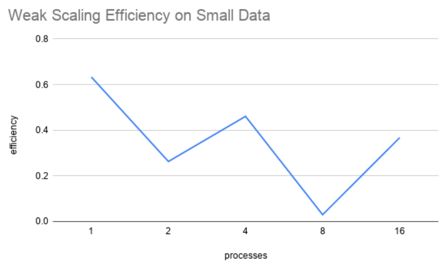


(b) Small Particle Sizes

Figure 4: Strong Scaling Efficiency



(a) Large Particle Sizes



(b) Small Particle Sizes

Figure 5: Weak Scaling Efficiency

computation aspect of the problem covers the effect of communication cost and actually benefits the performance much more.

2.4 Strong Scaling Efficiency

As discussed in the speedup section, the efficiency gain of the processes used is much more evident on some particular data size with appropriate number of processes. As Figure 4 shows, the data size of 80,000 on 8 and 16 number of processes provides greater computation load that greatly benefits runtime while it uses the best usage of the processes. The max efficiency is obtained on data size of 80,000 on 8 processes with value of 0.7. In constant, on smaller data sizes, the efficiency is greater when smaller number of processes is used and increase in number of processes lowers the efficiency usage of them. This is again because the number of computation done per process is not so much compared to communication.

2.5 Weak Scaling Efficiency

As discussed, desired efficiency of 1 is hard to be achieved when we are dealing with a problem that involves communication. As some of the data sizes provide more computation to communication ratio, we observe a better gain of efficiency for them . For instance in large data size of 80,000 with 2 or 8 number of processes, the max efficiency of 0.7 is achieved, while the best efficiency among small data sizes happen in size of 8,000 with 4 number of processes with value of 0.46. This is depicted in Figure 5.

Our implementation achieves the average efficiency of 0.41 on small data sizes, and average efficiency of 0.51 on large data sizes.

Overall the best results are obtained when we could make use of the time of processes by doing more computation and reduce idling time by doing a long communication specially if that is blocking. As indicated, we could have used the non-blocking functions of the MPI to overlap communication with computation and reduce the runtime much more.