CS267 HW2-2 Report

Gabriel Raulet and Yen-Hsiang Chang

1 Introduction

The goal of this homework is to design a parallel distributed-memory implementation of a particle simulator with `MPI`. In order to achieve reasonable scaling, we used a 2D rectangular processor grid to distribute particles in the square simulation space by their positions. This enables us to perform force computations and movement computations locally on each processor, with the caveat being that we must correctly and efficiently handle cases where particles interact (and move) across processor boundaries. Our approach builds on top of our previous shared-memory binning strategy, the major difference being that the bins are now rectangular to deal with the possibility that a non-square number of processors are used. Our evaluation for 6M particles shows that our distributed-memory implementation finishes in 17.986 seconds with 128 cores on Perlmutter.

2 Methodology

2.1 Processor grid

Our implementation uses a rectangular processor grid to distribute particles across processors. Let p be the total number of processors, and let $[0, M] \times [0, M]$ denote the simulation space in 2D Euclidean space. We then partition the square simulation space into p_r rows and p_c columns, where $p_r = \lfloor \sqrt{p} \rfloor$ and $p_c = \lfloor p/p_r \rfloor$, such that $p_r \cdot p_c \leq p$ and $p - p_r \cdot p_c$ is small in order to utilize most of the processors. A particle with position coordinates x and y would then be assigned to the processor row $\lfloor x/(M/p_r) \rfloor$ and processor column $\lfloor y/(M/p_c) \rfloor$, uniquely determining its assigned processor. Similarly, we subdivide each processor rectangle into a grid of bins, where the number of bins is chosen so that the smaller side of each rectangular bin is at least as big as the particle-interaction cutoff. Each processor also maintains an extra row and column (in each direction) of bins so that there is space to store bins communicated from adjacent processors.

2.2 Communicating ghost particles

In order to compute the acceleration vectors of a given particle, we need access to all particles in the (up to 9) adjacent neighboring bins. For bins that reside on the edge of the processor rectangle, we exchange bins with adjacent processors using non-blocking `MPI` send and receive calls, i.e., `Isend` and `Irecv`. This occurs in two stages: first the number of particles that will be needed for each adjacent processor is exchanged, and then these counts are used to exchange the proper number of particles.

2.3 Communicating boundary-traversing particles

The second place where communication occurs is after all the particles have been moved. Once each particle's position has been updated, we add all the particles that have crossed their assigned processor boundary to a buffer and send them to their proper destinations using `Alltoall` and `Alltoallv` collectives. The reason why we use all-to-all collectives is that we do not rely on the assumption that particles cannot move more than one processor away.

2.4 Gathering particles to the root

We also have to perform communication in order to gather all particles to the root processor in order to write our results to disk. Because the proper ordering of particles will have been significantly shuffled throughout the simulation, we store each particle in a structure containing its original global index and gather them to the root

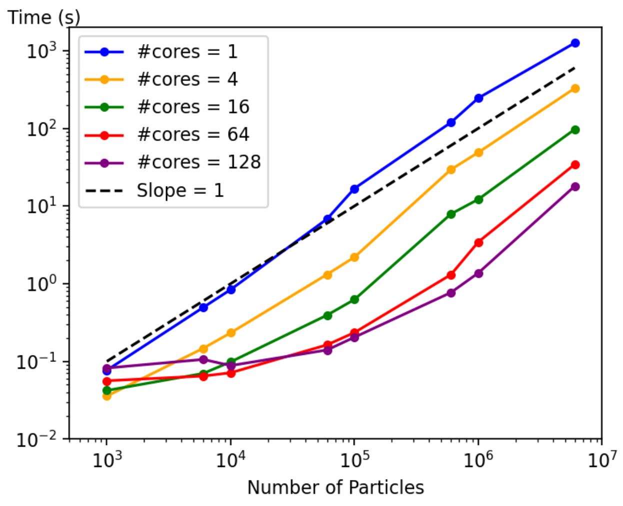

Figure 1: Experiments showing linear complexity.

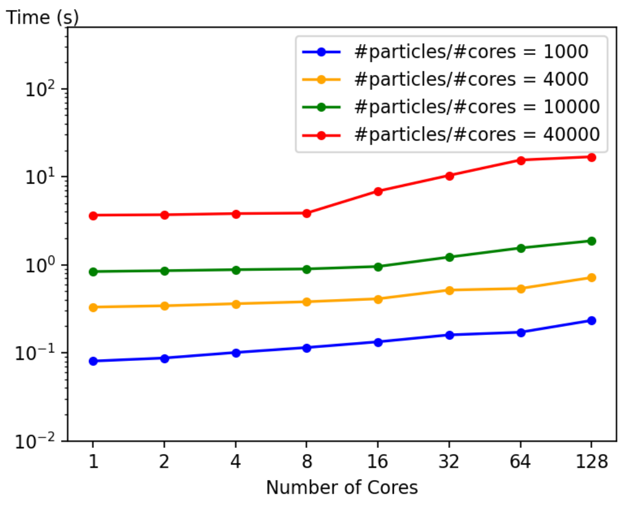

Figure 2: Weak scaling experiments.

with a custom `MPI` datatype. Once all particles have been gathered, the proper global ordering is quickly achieved using the coupled global indices.

2.5 Design choices

Our first implementation used a virtual graph topology to communicate particles between adjacent processors using Neighborhood collectives. A benefit of this approach is that it simplifies the communication of ghost particles into just two collectives calls, one for obtaining the number of particles to exchange and one for doing the actual particle exchange. A downside of our particular implementation is that it required re-binning all the ghost particles on the received processor. It requires synchronizing all processors, even when some processors may have many fewer particles to exchange than others. Using non-blocking send and receive calls, we are able to send particles straight to their corresponding ghost bins. Furthermore, processors that don't need to exchange a lot of particles can move on to performing force computations for particles not on boundaries without having to wait for every other processor to finish communicating.

3 Experiments

3.1 Linear Complexity

Figure 1 demonstrates how our distributed implementation scales when we increment the problem size from 1K to 6M. By following the techniques described in Section 2, our solution has slopes close to 1 in the log-log plot whenever the number of particles is not less than 100K, indicating that our solution is indeed linear. It's normal that our solution does not show a linear trend when the problem size is small and the number of cores is large, since there might not be enough work that can be distributed across cores, and the overhead for calling functions and initialization dominates computation time.

3.2 Weak Scaling

Figure 2 shows our weak scaling results, where we fix the ratio between the problem size and the number of cores to 1K, 4K, 10K and 40K, respectively. As we can see, our solution scales well up to 8 cores, and slightly moves away from perfect scaling (slope of 0) starting from 16 cores.

3.3 Strong Scaling

Figure 3 shows our strong scaling results, where we fix the problem size to 1K, 10K, 100K and 1M, and 6M, respectively. As we can see, our solution scales well when the problem size is large enough, where our solution has slopes close to -1 in the log-log plot.

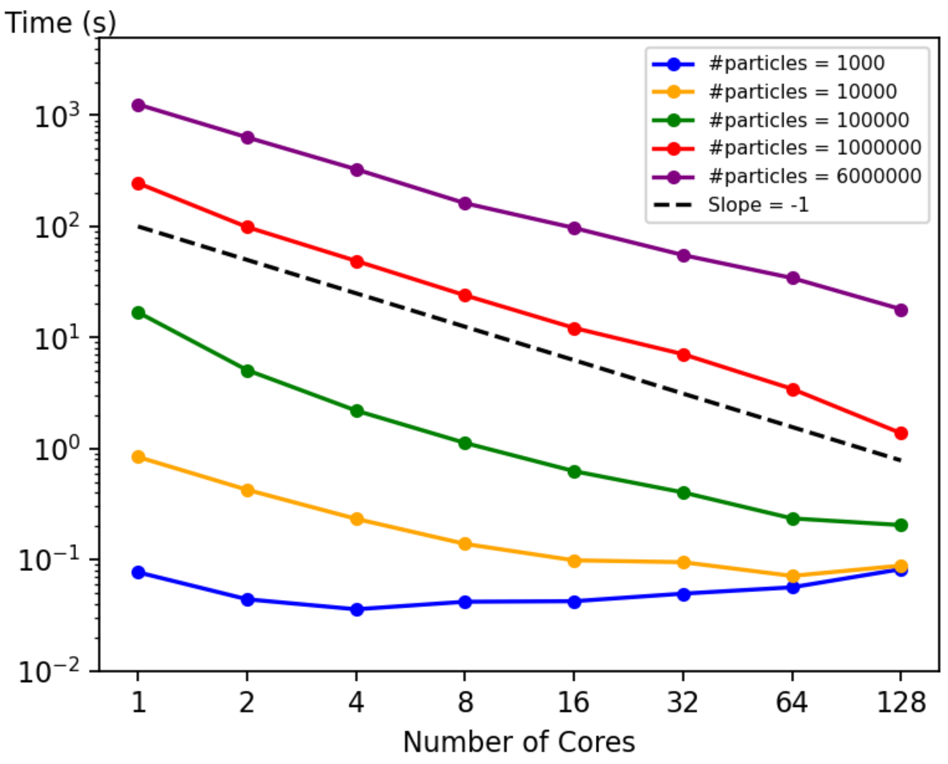

Figure 3: Strong scaling experiments.

Table 1: Time breakdown.

#cores	Computation (s)	Communication (Isend/Irecv) (s)	Communication (Alltoall/Alltoallv) (s)
1	201.98	N/A	N/A
2	100.019	0.115	0.119
4	51.781	0.142	0.082
8	23.765	0.175	0.062
16	12.343	0.095	0.052
32	6.650	0.093	0.056
64	3.184	0.099	0.062
128	1.064	0.130	0.088

3.4 Time Breakdown

To understand what is going on inside our implementations, we decide to break down the running time into computation time, communication time for `Isend/Irecv`, and communication time for `Alltoall/Alltoallv`. We focus on the data set of 1M particles and investigate the breakdown for different number of cores.

Table 1 shows the time breakdown results. Based on this experiment, we observe that the running time is dominated by the computation time. We also observe that, for the communication time, the time spent on communicating ghost particles is the main communication bottleneck. This makes sense due to the fact that many more ghost particles are communicated at each simulation step compared to the number of particles that change processors. We were somewhat surprised that the `Alltoall` communication phase seemed to relatively small in comparison. There is probably a good reason for this: most particles don't change processors at a given simulation step. We hypothesize that this communication phase could become a bottleneck at much higher processor counts, in which case a solution based on neighborhood collectives would then be desirable.

4 Conclusion

In this homework, we implemented a scalable distributed-memory solution for a low-density particle simulation. Using a 2D processor grid, we are able to significantly reduce the amount of communication needed to perform force computations. There is room for improvement in our handling of boundary-traversing particles. Instead of using `Alltoall` collectives, an approach based on our earlier attempt with neighborhood collectives would likely have made our solution more scalable, as the number of possible destination processors for each moved particle would go from $O(p)$ to $O(1)$.

Contributions

Both authors contributed equally to this report.