Parallelizing a Particle Simulation - MPI

Tongge Wu[1], Debbie Liang[2], Zhuoming Zhang[2]

[1] *Department of Mechanical Engineering, University of California, Berkeley*
[2] *Department of EECS, University of California, Berkeley*
March 1st, 2020

1 Introduction

Molecular dynamics simulation, as a well-studied problem in the scientific computing field, is widely used in biochemistry, mechanics, and astrophysics. The basic problem setting is as follows:

- Given a domain (i.e., square) and boundary conditions (i.e., fully elastic recover).
- Initialize positions and velocities of a group of particles with prescribed masses.
- The force between any pair of particles is a prescribed function of the distance between the two particles.
- All particles move within the domain according to Newton's second law in a finite time interval.

In this assignment, we optimized a naive molecular dynamic simulation on a distributed-memory machine using Message Passing Interface (MPI). Here was our contribution:

- Tongge Wu worked on implementing the gathering function that collects the results of simulation from multiple processors, testing scalability, providing mathematical support to our algorithms and writing the report
- Debbie Liang worked on implementing the function to move particles, sending & receiving particles, and repartition received particles, visualizing data, and writing the report
- Zhuoming Zhang worked on implementing the function to simulate one step, incorporating code to fit the theoretical data structure, testing correctness, and working on timing analysis using HPCToolkit, and writing the report

2 MPI Framework

2.1 Cell List Methodology

To achieve $\mathcal{O}(n)$ run-time, a constant amount of computation for each particle was needed. We adopted the idea of *Cell Lists*, which makes each MPI rank to be in charge of a sub-domain and all the particles within the sub-domain [1]. We further divided each sub-domain into cells, whose side length was around the cut-off distance in the force function. Thus, a "vector<vector<list<int> > >" data structure was constructed to represent the map between cell position and particle I.D. within one rank. Furthermore, it was necessary to update the map after each time step, since particle might leave one sub-domain and enter another sub-domain during the entire simulation.

To compute the resultant force applied on each particle, we first located the cell that this particle belonged to. Then, we looped all particles in the center cell and 8 neighbor cells to compute interaction between each pair of particles. Since this particular particle could potentially generate force with only particles in the center cell, as well as particles in the 8 neighbor cells.

When all particles' forces, and hence, accelerations were calculated, we would move the particles for a given time step using the Verlet ODE integrator. After all particles' positions were updated, each rank needed to communicate with the "neighbor" ranks (ranks that in charge of physical neighbor sub-domain) to swap particles that went beyond the boundaries (boundaries between each physical neighbor sub-domain). By the end of the communication, all computation within one time step was done, the next time step could start if it was not the end of the simulation.

3 MPI Optimization

3.1 Locality Efficiency

The rationale behind our design (Cell List) was that particles were likely to stay in a group of cells that were adjacent to each other. If each rank took care of a group of cells, it was more likely that a particle position would be updated by the same rank for a few time steps. Hence, it reduced communication between ranks.

To achieve such locality efficiency, we came up with two designs to partition the whole domain: 1-d partition and 2-d partition.
1-d partition was to partition the domain horizontally into short fat rows. While 2-d partition was further partitioned

the short fat rows vertically so each rank would be in charge of a square sub-domain.

We went with the 1-d partition design considering following reasons: Firstly, The 1-d partition required less communication (`send` and `recv` calls) than the 2-d partition. Since each rank only needs to communicate with the rank above and the below it.
While for the 2-d partition, each rank needs to communicate with the eight neighbor ranks. Therefore, it could reduce the latency cost. Secondly, the package (a group of particles that went beyond the boundary) sent from one rank to another rank within one time step was not large. Averagely, each rank would send 25 particles to other ranks for a regular scale simulation ($n = 1000$ particles) at one time step, where the simulation was done with 10 MPI ranks. Since the model of communication in MPI was described as

$$\alpha + n\beta$$

where α is the latency cost, and β is 1 over bandwidth. Thus, the latency cost would dominate the communication cost in most of our simulations.

3.2 Redundant Storage and Computation

To further reduce the communication overhead, we use the overlap technique between each rank, as shown in Fig.1.

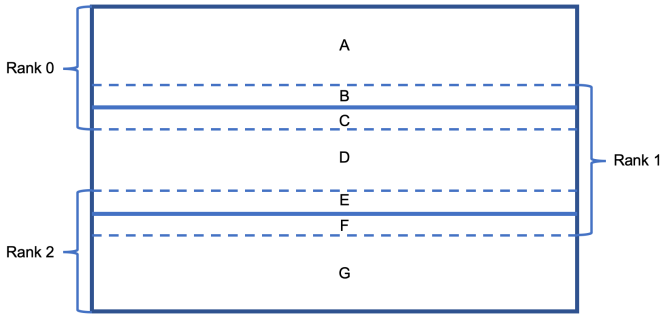

Figure 1: Rank 0 was responsible for applying force and moving particles in region A and B, but it also has the map of region C in its memory; Rank 1 was responsible for applying force and moving particles in region C, D, and E, but it also has the map of region F in its memory; Rank 2 was responsible for applying force and moving particles in region F and G, but it also had the map region E in its memory.

In the applying force section, rank 0, for example, would calculate the resultant force of particles in A and B regions. To calculate the resultant force of particles in the B region, rank 0 would need positions of all particles in the region C. Since rank 0 had a map of the region C in its memory, there was no need to communicate with rank 1 to get that information. In the moving section, rank 0, for example, would update the position of each particle in the region A and B.

After the moving section, rank 1, for example, would send a list of particles moved from C to B (package δb) and a list of particles in the region C (package c) to rank 0. Rank 1 would also send a list of particles moved from E to F (package δf) and a list of particle in region E (package e)to rank 2. Symmetrically, rank 0 would send a list of particle moved from B to C (package δc) and a list of particle in region B (package b) to rank 1; rank 2 would send a list of particle moved from F to E (package δe) and a list of particle in region F (package f) to rank 1.

In the unpacking section, rank 1, for example, would combine c and δc to get a total update of the region C, meanwhile, rank 1 would also combine b and δb to get a total update of the region B. Redundantly, rank 0 would do the exact same work as rank 1 to get total updates of the region C and B. Thus, in the next time step, all ranks had the necessary information to compute the resultant force.

If we implement the sub-domain in an alternative way such that there was no overlap of sub-domain between each rank, applying force would require a round of `send` and `recv`, thus, increased the cost of latency.

3.3 Physical Foundation

In our code, we only considered the situations, for example, when particles in the region B moving to region C but not from region B to region D within one time step, considering region B and C had one cut-off length as their height. We were concerned about how to deal with particles flying over a very long distance within one time step. However, according to the number given in `common.h`, each time step was $0.0005s$ and the cut-off distance was $0.01m$. And the main function initialized the horizontal and vertical velocity of each particle between $-1m/s$ and $1m/s$. Thus, the magnitude of the initial velocity of each particle is $\sqrt{2}m/s$ at most.
Physically, the whole domain containing all particles was initialized in a thermodynamic equilibrium status. According to the Second Law of Thermodynamics, such a system would not have temperature differences within the region spontaneously. Thus, we concluded that the chance that a particle has a velocity magnitude larger than $\sqrt{2}m/s$ is minimal.

4 MPI Communication

4.1 Packed Particles

To reduce the number of calls to `send` and `recv`, therefore the cost of latency, we packed all the particles together in each region to send by each rank. Moreover, increasing the size of the package would also increase the bandwidth, considering that each particle was just 56 bytes.

4.2 Packed Partitions

Another technique we used to reduce the `send` and `recv` calls was pack the package of two regions. Take rank 1 as an example, we packed the package δb and package c into a larger package, then send it to rank 0 as a whole. After receiving the whole package, rank 0 would differentiate the particles based on their positions. This redundant computation saved time of communication by reducing the calls to `send` and `recv`.

4.3 Synchronization

4.3.1 Blocking Send/Receive

In the previous section, we discussed that each rank needed to send two packages and to receive two packages. In our first implementation, we uses interleaving blocking `send` and `recv`. In each time step, each even rank would send, receive, send, and then receive again, and each odd rank would receive, send, receive, and send. These operations avoided deadlock caused by blocking message passing. Furthermore, rank 0 and last rank only needed to perform one sending and receiving operation as they only needed to communicate with one rank. A barrier was added at the end of `simulate-one-step` function to make sure all ranks were synchronized based on the step.

4.3.2 Non-Blocking Send/Receive

We considered using non-blocking `send` and `recv` could reduce MPI overhead, and it was an easy way to avoid deadlock caused by blocking communication. Thus, we changed from blocking to non-blocking `send` and `recv` using `mpi_isend` and `mpi_irecv`. The implementation turned out to be straightforward because we did not have to interleave communication based on even and odd. Each rank issued two sending operations, one to rank - 1, and another one to rank + 1. Similarly, each rank issued two receiving operations, one from rank - 1 another from rank + 1. Rank 0 and last rank only had one pair of send and receive because they are adjacent to one rank instead of two ranks. Finally, we use `waitall` function to ensure all ranks send and receive all messages they needed. `waitall` could also synchronize every rank so that the step number of messages was the same for every rank. Non-blocking communication was approximately 10% faster than the blocking communication method.

5 Additional General Optimization

To iterate through the 8 cells surrounding the current particle, we first created a vector that held the 9 cells and used a for loop to iterate. In the process of optimizing our code, we discovered that unrolling the `for` loop to be 9 similar blocks of code improves the run time. The improvement may due to minimizing the branch prediction. In conclusion, unrolling sped up our code by up to a factor of 3.

6 MPI Performance

All results were checked with the naive (serial) particle simulation ($\mathcal{O}(n^2)$) solution [1].

6.1 Complexity with Simulation Scale

When we fixed the number of ranks to be 68 on 1 computing node (and on 2 computing nodes) and progressively scaled up the simulation, the computation time was collected and listed in Table.1 and Table.2.

A linear regression analysis showed that the complexity was $\mathcal{O}(n)$ (Eq.1). The data was also shown in Fig.2. We found that, with the same number of ranks, using 2 nodes would increase the speed by 10%. It indicated that with multiple nodes, we had access to more caches, which allowed the function to have faster memory read and write time.

Number of particles	Wall-time(s)
50K	0.461151
100K	1.1082
200K	2.32307
400K	4.54783
800K	9.63563
1M	12.1667
2M	25.5338
4M	49.7345
8M	105.882
10M	134.799
20M	288.975

Table 1: Scalability testing: relation between simulation scale (K: $\times 10^3$; M: $\times 10^6$) and computation time for total of 68 ranks on 1 node

Number of particles	Wall-time(s)
50K	0.68855
100K	1.20377
200K	2.19192
400K	4.3707
800K	8.69291
1M	10.3756
2M	21.5452
4M	43.7266
8M	94.1803
10M	123.164
20M	262.55
40M	549.158

Table 2: Scalability testing: relation between simulation scale (K: $\times 10^3$; M: $\times 10^6$) and computation time for total of 68 ranks on 2 nodes

$$\log(T) = 1.06\log(n) - 12.11 \quad \text{with} \quad R^2 = 0.999 \quad (1)$$

[1] We had a particular issue with 5000 particle simulation with random seed of 1, we had average distance result 1.01e-7 for first 500 step and 0.012 for 1000 steps. All other seed's error are within half of the threshold

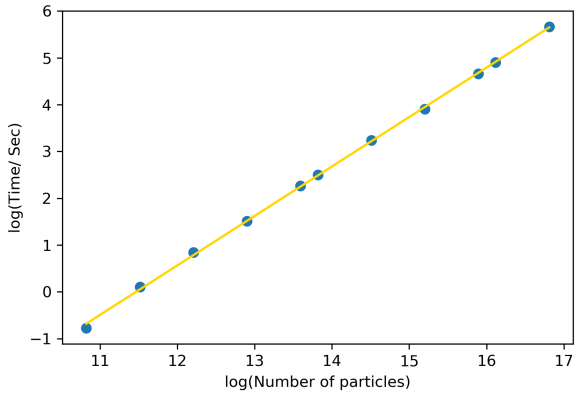

Figure 2: Scalability testing: computational time as a function of simulation scale. Blue dots are numerical experiment results, while the yellow line is the linear regression of these data points. All data points are log transformed from Table.1 with 1 node

6.2 Weak scaling

To obtain the weak scaling characteristic, we assigned each rank with 150K particles, then progressively increased the number of ranks. The computation time was in Table.3 and Table.4.

If we define the Efficiency as

$$\text{Efficiency} = \frac{\text{Time(1 rank)}}{\text{Time(p ranks)}}$$

we could plot the efficiency as a function of number of ranks, see Fig.3. We observed that our efficiency decreased when the number of ranks increased. It indicated that an increasing number of rank increased overhead of communication. The overhead of communication grew gradually, and we also obtained a much better weak scale than OpenMP.

Number of ranks	Number of particles	Wall-time(s)
1	150K	93.1021
2	300K	95.7631
4	600K	96.761
8	1.2M	101.1
16	2.4M	106.424
32	4.8M	113.732
64	9.6M	139.173
68	10.2M	137.509

Table 3: Weak scaling: relation between simulation scale and computation time for fixed number particle per rank (150K) for 1 node

6.3 Strong Scaling

To obtain strong scaling characteristics, we fixed the simulation scale (10M particles) and progressively increased the number of ranks. The computation time was in Table.5.

Total Number of ranks	Number of particles	Wall-time(s)
2	300K	95.9879
4	600K	97.4501
8	1.2M	99.3473
16	2.4M	104.992
32	4.8M	113.946
64	9.6M	126.38
128	19.2M	149.299

Table 4: Weak scaling: relation between simulation scale and computation time for fixed number particle per rank (150K) for 2 nodes

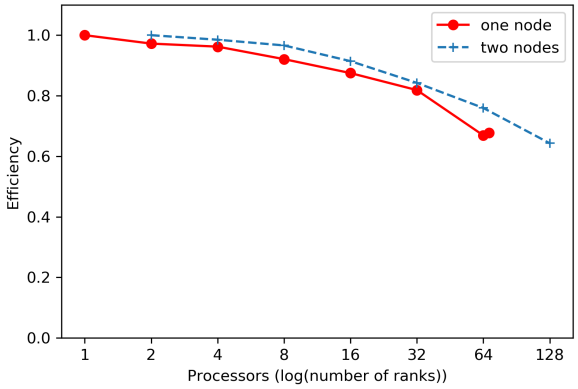

Figure 3: Weak Scaling: efficiency as a function of number of ranks. Red dots are data from Table.3, while blue dots are data from Table.4

If we define the Speedup as

$$\text{Speedup} = \frac{\text{Time(p rank)}}{\text{Time(1 rank)}}$$

we could plot the speedup against the number of ranks, see Fig.4. The plot suggested we had a good strong linear scale from the MPI code. However, we could not be able to reach the ideal case due to communication overhead when the number of ranks was above 64. The reason was that each rank needed to send and receive messages about particles in the ghost zone to proceed. When the number of ranks increased, the latency of each communication increased. One possible way to minimize the communication between rank was to change the partition to the 2D grid instead of 1D roll, but the implementing difficulty also increased.

6.4 Runtime Breakdown

To obtain the timing analysis of our MPI code, we used HPCToolKit installed on Cori. The communication time was the summation of all wait, isend, irecv operations. Basically, We fixed the simulation scale to be 1M particles and progressively increased the number of ranks, see Fig.5. As shown in the figure, we observed that the synchronization time fraction decreased initially and increased significantly when the number of ranks was over 32. To explain this behavior, we thought that when we requested 4 ranks in MPI, the communication was

Number of ranks	Wall-time(s)
1	7704.08
2	3705.64
4	1941.36
8	990.66
16	489.574
32	242.186
64	144.17
68	139.487
64*	130.978
128*	76.6184
256*	52.5238
256**	40.6055
512**	28.8605

Table 5: Strong scaling: relation between number of ranks (most cases on 1-node, $*$: 2-node, $**$: 4-node) and computation time for fixed number particle (10M).

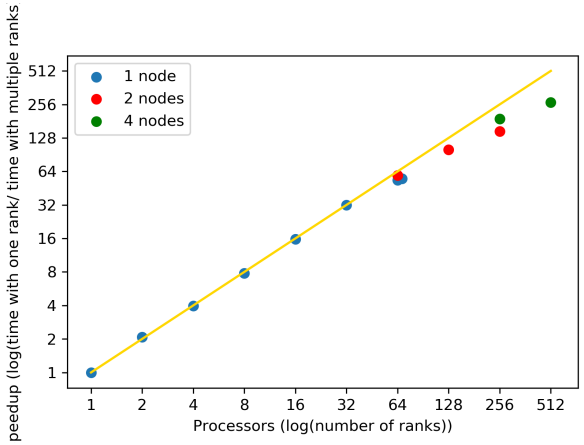

Figure 4: Strong scaling: Speedup as a function of number of ranks for a fixed simulation scale. The golden line indicates the ideal speedup with slope of one. Blue dots are one-node data, red dots are two-node data, green dots are four-node data in Table.5.

low because 2 of the ranks only needed to send and receive once, while the other two ranks needed 2 send and 2 receive. As the number of ranks increased, the total amount of send and receive operation increased twice as fast. Thus, the communication time increased much faster than the number of ranks.

7 Conclusion

In this homework, we designed and optimized parallel molecular dynamic simulation using MPI. We achieved $\mathrm{O}(n)$ complexity using MPI. Moreover, the weak scaling test showed that the performance of each rank decreases gradually as the rank increased, achieving a better result than the OpenMP code. At the same time, the strong scaling test showed our code linearly scaled up to 512 ranks (using 4 nodes), which was optimal on computing nodes on Cori. In conclusion, we achieved a reasonable parallel version of the basic molecular

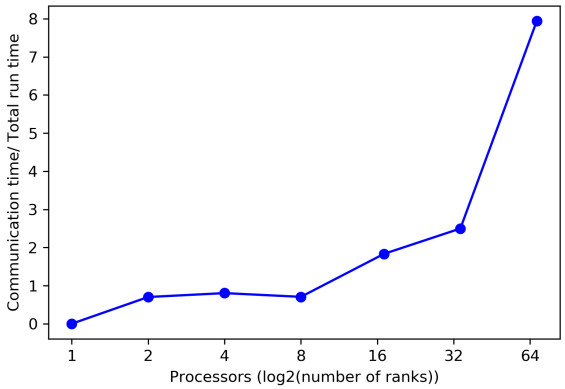

Figure 5: Runtime Analysis: Communication time fraction as a function number of ranks, given a fixed simulation scale (1M particles)

dynamic simulation.

References

[1] Victor Eijkhout with Robert van de Geijn and Edmond Chow. *Introduction to High Performance Scientific Computing*. http://www.tacc.utexas.edu/~eijkhout/istc/istc.html. lulu.com, 2011. ISBN: 978-1-257-99254-6.