CS 267 - HW2.2 (Group 22)

Akul Arora* Dhanush Nanjunda Reddy†
Pedro Guimarães Martins‡

March 4, 2023

1 Methods

1.1 Parallelization by Bins

In order to implement a parallel version of this molecular dynamics code, we have to split the simulation cell among many MPI processors. Here, our starting point is the serial code implemented in homework 2-1, in which we implemented a binning approach to make the non-parallel code scale linearly with the number of particles. Therefore, now, there is a number of possible designs on how to split these bins accross many MPI processors.

One possibility is assigning a number of rows to each processor, which leads to a relatively simpler implementation, but involves extensive communication between processors. This is the case because all particles sitting in bins that fall on the border of a processor's region will be crucial for force computations in different processors, and, therefore, will have to be communicated. Another possibility is splitting m x n blocks of bins to each processor, which despite being more complex, is able to define regions with lower perimeter:area ratio, which means we are able to perform less communication between processors.

We attempted to implement both approaches, but only the 2D Bins passed the correctness test, and is ultimately our final submission. In any case, here we will include the design and algorithmic choices for both approaches.

1.1.1 1D Row Parallelization

To implement a 1D binning scheme, we first partitioned all the rows in the system across processors in the init_simulation function, with some processor taking additional leftover rows as the number of rows might not be a multiple of the number of processors. Then, we assigned the corresponding bin indices to each processor; finally, we looped over the particles, populating a vector of vectors of particle-like objects locally in each processor if the particle belonged

*Akul (akularora@berkeley.edu) worked on implementing mpi.cpp and writing the final report.
†Dhanush (dhanushn@berkeley.edu) worked on implementing mpi.cpp and writing the final report.
‡Pedro (pedrogm@berkeley.edu) worked on implementing mpi.cpp and writing the final report.

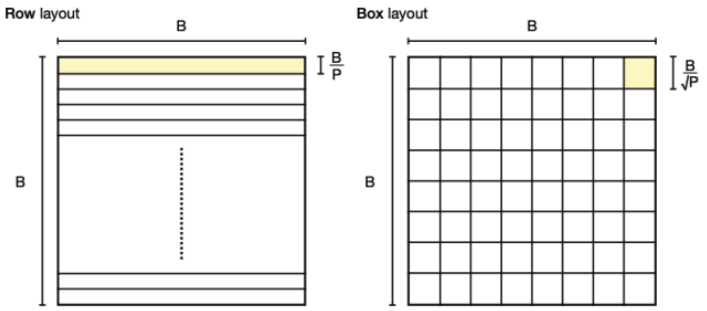

Figure 1: Figure 1: Left is Row Layout (1D) and Right is Box Layout (2D). Figure taken from recitation.

to any of its assigned bins. We also allocated memory to a couple of arrays used later to manage re-binning particles.

Upon splitting the simulation box among processors, it was important to consider how to calculate forces for particles in bins adjacent to the border of the processor assigned region. As we repeat our serial strategy of only calculating forces between particles sitting in adjacent bins (either sharing a bin wall or bin corner), it is important that each processor also has access to the particles in bins immediately above and below its top and bottom row respectively. Consequently, we had to implement a communication scheme between MPI processors, which was done in the simulate_one_step function.

First, we needed to check whether the processor did share a top or bottom border with another processor, as the processor dealing with the first or last rows would not need additional "ghost" particles to compute forces in bins sitting in either of these rows. After that was done, we needed to identify the row number of its top and bottom rows, collect all the bins and particles in this region, and finally calculate the rank of the processor that would receive these particles. Once that was done, we used MPI_Isend to send this information out. Concurrently, we had to implement a receiving scheme for each processor. We started with a buffer vector to receive all the particles, and then looped over them and assigned them to the correct bins in this new processor, populating locally the vector of vectors bins.

Once that was done, we were finally able to compute forces, looping over each bin in the processor, and then over each of the particles sitting there. Then, we checked whether this bin has bordering bins, since corner and border bins in the simulation box might not have immediately contiguous bins in some of the eight surrounding positions. After going over all the existing neighboring bins' particles, we also looped over the particles in this same bin, of course, excluding self-interaction. Once that was done, we deleted all the communication objects, so they are clear for another round of communicating ghost particles in the next step, and finally moved the particles according to the calculated forces.

The last important step in this function was to rebin particles that might

have moved to a different bin or even to a region which corresponds to another processor all together. In this process, we had to loop over all the bins assigned to this processor and identify all particles with bin assignment differing from its current one. Once all the MPI processors reach this point, as we implemented a barrier, we can gather all the misplaced particles in a single vector using MPI_Allgather and MPI_Allgatherv. First, we gather the number of particles misplaced in each processor and record their incremental cumulative sum, as we need to inform each process the address in the final vector it should start writing to avoid any overwriting. Once this is done, each processor iterated over this list of particles and copied in the appropriate bins any particle that might belong to its list of bins.

This approach ultimately showed to be fairly limited in terms of efficiency. First, when communicating ghost particles between processors, we could have sent not only the particles but also their bin assignment, so that calculation would not have to be done once again when the particle is received. Another important source of improvement could be when re-bining particles, since we ultimately computed bins twice, first to identify the misplaced particles, and then second to assign them to the correct bins. In a more efficient implementation, we could not only have transferred the misplaced particles but also their new bin assignment, so as to trade some additional communication cost for significantly reducing computations. Finally, for this last re-binning step, it could also be more efficient to use MPI_Send and MPI_Receive, and directly give to the correct processor all the particles that belong to their bins. This would also avoid having every single processor searching for particles in a unique misplaced particles vector.

Ultimately, we were not able to achieve correctness with this code, as very small deviations in particle position upon the first few dozen steps would sequentially accumulate, and the final trajectories would deviate significantly from the expected results. We focused then on implementing a 2D binning scheme, which can decrease communication between processors and dramatically increase efficiency, which we were able to fully implement and achieve correctness.

1.1.2 2D Box Layout Parallelization

Our second approach to decomposing and assigning the bins into processes was to create a box layout where each process was responsible for a square box of bins in the grid of bins. This approach was more challenging to implement, but provides better results for the following reason.

This approach ends up communicating less data as we only need information from neighboring boxes' processes, which share a smaller border with the current box than with the row layout. This results in requiring less data to be shared across processes as there is less information to be shared across a smaller border. We verified this empirically by printing the number of particles that we were sharing in each simulation step.

One issue that came up in implementation was dealing with a non-perfect square number of processes. Our solution to this was to simply pick the first

perfect square below the number of processes provided and use that number of processes instead. This made implementation far simpler, albeit at the cost of slightly more optimized performance by leveraging closer to the full number of processors available.

This was our final choice for our implementation due to the reduced communication needs. We address how we implemented communication for this approach in the following section.

Regarding data structures, we stored the bins relevant to that process in a vector of vectors, which provides a grid-like structure in memory. We also stored an additional edge to all the bins due to ghost particles, which is explained below. We used a vector of vectors that are of type particle_t (not the pointer). We chose to store the actual particles in the vector rather than the pointers to the vector to avoid issues with improper memory accesses and sharing data that would be caused by passes by reference rather than by value. Storing the particles themselves took up more memory, but made the implementation far simpler. Since this task isn't very memory-intensive this wasn't a concern.

1.2 Communication

1.2.1 Communication Methods

Initial Methods Attempted.

In addition to trying the gather method for sharing data, we also tried using Alltoall communication as well as manually sharing data between processors using Send, Receive, and Probe.

Alltoall(v) communication allows for a process to send unique data to each individual process and repeat this for all processes. This is ideal for our application since it allows for processors to only send data to other relevant processors. This way we are minimizing communication by only communicating data that is necessary for the processor to proceed correctly. This approach is also ideal as it allows for us to specify what we want to send where and let MPI handle how to optimally communicate this information. While this implementation approach is ideal, we ran into a number of implementation issues that were difficult to debug.

To remediate the implementation issues, we tried manually writing the code for sending and receiving between processes using the basic Send and Receive commands. This approach would have been similar to Alltoall since we were sending data to all neighboring processes that required the specified particles. Here again we unfortunately ran into implementation issues. This approach was particularly convoluted as we had to do a lot of work normally handled by MPI ourselves.

In the end, we went with the Gather approach and saw good results.

Gather Communication.

Our final implementation attempt was using Gather and AllGather. Gather allows us to send data from all processes to a single process (useful for gather_for_save())

and AllGather allows us to send data from all process to all processes (useful for simulate_one_step()).

This communication approach is more inefficient compared to the previous methods attempted since they have a process sending data to all other processes, instead of just the neighboring processes that require the data. On the whole though, we realized this is still relatively cheap since a box layout requires a process to send very little data as it is. Once all processors had received all shared particles, they would go through and check to see if the particle shared was meant for them, and if so, process the particle.

This approach proved to be only slightly less efficient, but far simpler to implement. We thus went with this approach in our final implementation.

Regarding data structures, we used a vector to store the particles that were to be sent over and then used a pointer to receive the buffer of particles sent over. The vector made it simple to easily add and remove particles without having to reallocate memory on the heap. We also had the vector hold copies of the particles instead of pointers to the particles, so we could avoid issues with passing by reference and improper memory accesses when sending and receiving items.

This was our final choice for our implementation.

1.2.2 Communication Approach Between Processors

Communication requires handling ghost particles.

In order to apply forces to each particle, we must consider all of the particles in the same bin as the particle as well as all neighboring bins. This is the case since we set size of our bins to be the cutoff size = 0.01. Since a particle can only be affected by particles at most cutoff distance away, we are constrained to only the current and neighboring bins.

This produces an issue when we are applying forces to particles in a bin that is on the edge of the processor's range of bins that it is responsible for. Some of the neighboring bins will be managed by a different process. We thus need to communicate these "ghost particles" to the current process.

Rather than communicating these particles while applying forces, which would cause multiple communications in the same step, we choose to copy the ghost particles into the bins space that the current processor is handling. We choose to do the latter since doing the former requires a larger amount of communication and makes implementation more confusing. We do this by creating an extra edge of surrounding bins in the processor's bin space that is reserved for ghost bins. Prior to applying forces, we copy these ghost bins into the current processor's bin space. We then proceed as normal.

Note that one thing we could have improved on was that at the start of each step, we will clear all ghost bins and then re-communicate them. It would have been more efficient to keep particles that got moved out of the current processor, but we knew they were going to be ghost particles for this processor or ghost particles that didn't move. We chose not to implement this though as it makes the code significantly more complicated and could have caused other

issues. We also observed that with the number of ghost particles being moved, this does not create significant additional communication overhead.

Communication distance is at most one processor away.

As you can see in Figure 2, for a box with 1,000 particles, forces decay exponentially with distance, and ultimately only short-range interactions will significantly affect the dynamics of this system. In this context, we followed a binning strategy, in which for each step we sequentially visit a spatial partition of the space - or bin, and compute pair-wise interactions between its particles and the ones sitting in neighboring partitions.

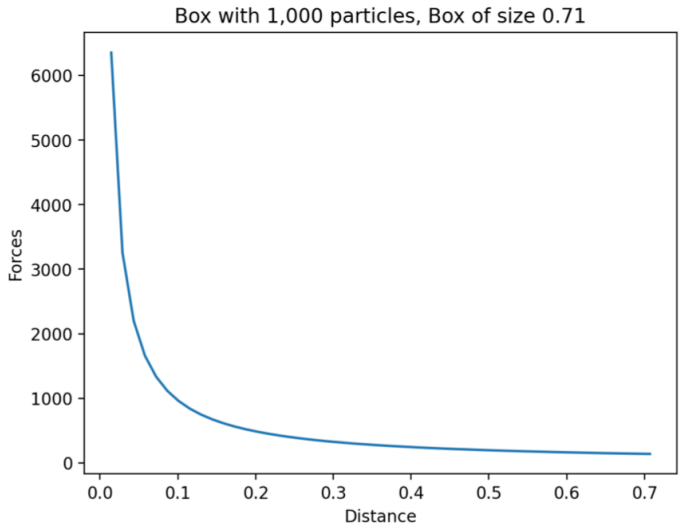

Figure 2: Figure 1: Particle-particle interaction potential

This distance limit can also be verified mathematically by evaluating the density and estimating the probability of movement to different bins based upon the different forces acting upon a particle. It can be shown that the probability that a particle will move more than one bin away (for the chosen bin size) is practically zero. The key takeaway from either approach is that there is a limit on the realistic distance that a particle can move during the simulation. This provides us with a neat constraint on the communication distance.

Our bin size is the cutoff size = 0.01, which is the range in which another particle can affect this particle and each process is responsible for # of bins / sqrt(# of processes) squared. So, with a sufficiently large number of particles, we can ensure that particles will move at most one processor away. Hence, we can limit communication to all neighboring processes.

We verify this with correctness tests to ensure we get the correct results with this assumption and confirm that this is true.

2 Results

2.1 Linear Scaling - MPI Implementation

First, we wanted to make sure that we were still recovering at least O(n) linear scaling with number of particles. As you can see in Figure 3, up to 6 million particles, the code proposed is robust, and the total time it takes does not increase more than linearly on average. **For 6 million particles particularly, our implementation took around 18 seconds**. We do see a dip in performance at around 600k particles, which most likely is related to memory access issues; as there are many layers of cache memory, at some point we might exhaust a more accessible level and have to resort to communication with lower latency/bandwidth. As a consquence, the overall gains in efficiency are not uniform, as they can be larger or smaller depending on the memory access changes as we simulate with more and more particles.

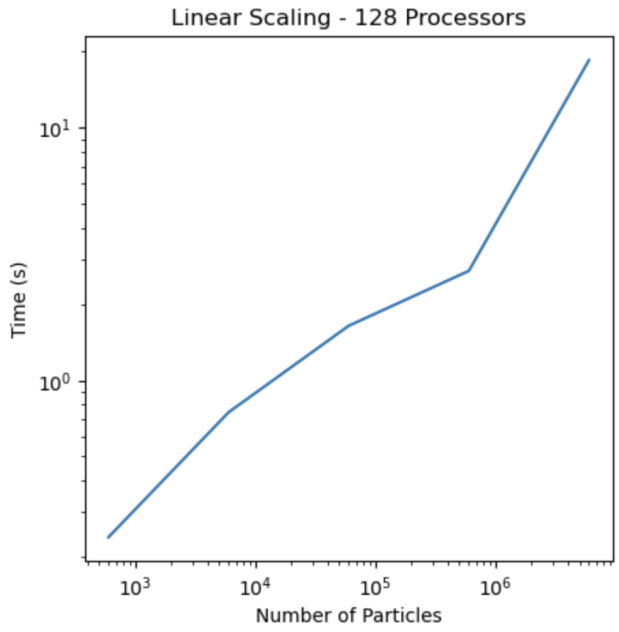

Figure 3: Linear scaling with 2 nodes, 64 processors each

2.2 Strong and Weak Scaling

In Figure 4, you can see the graphs for weak and strong scaling for this system. Overall, our code showed close to ideal behavior generally; in weak scaling the execution time for 2 or 64 processors was similar when scaling accordingly the

number of particles, while for strong scaling it was continuously below the line of slope -1, in black.

Nonetheless, we detect strong instabilities in our code, with some additions of processors resorting in great gains in efficiency, while other showed basically no improvement at all. For instance, between 4 and 8 processors, the strong scaling curve is practically flat, as well as between 32 and 64. If we had access to a CPU MPI activity profiling tool in Perlmutter, we would be able to understand exactly what might be happening here (time analysis performed in Cori follows in a later section). Nonetheless, one possible reason for this behavior is related to the trade-off between efficiency and communication as the particles are further spread among many processors. It seems that as we expand from 4 to 8 processors, for instance, the efficiency gain from now having 8 partitions of space simultaneously computing does not compensate the now additional costs of the rapidly scaling communication between the processors. Similarly, as we go from 32 to 64 processors, theoretically we have twice the computing power, but once again, the additional costs with communication even out the gains, and ultimately the execution time remains flat. Another explanation for this trend is related to how we utilize processors in the implementation. Our code is optimized to work with a perfect square number of processors, so it is only able to see significant improvement if the expansion in processors leads to at least another perfect square number. So for instance, as we go from 16 to 32 processors, essentially we are comparing the performance of 16 and 25 processors, which explains that the improvement does not scale as much as expected. Still, when we go from 16 to 64 processors, when the processor utilization is full, we can see both weak and strong scaling holding close to ideal behavior.

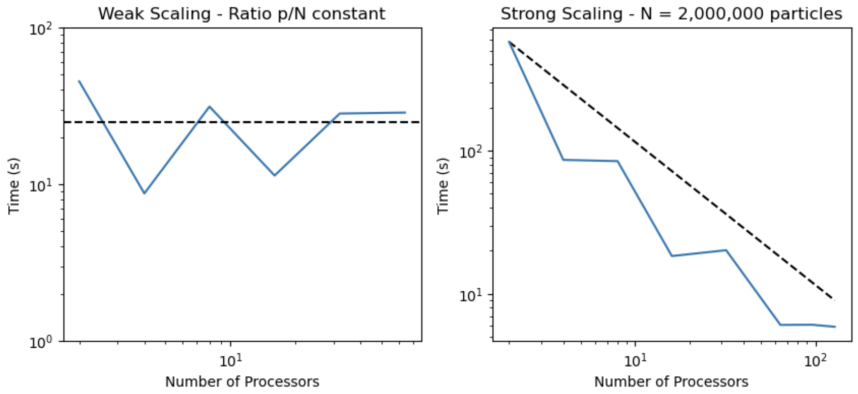

Figure 4: Strong and Weak scaling in Perlmutter

2.3 Synchronization and Computation Time Analysis

In order to investigate the synchronization and computation time, we used the vtune package implemented in Cori. Once again, we were not able to find a easy application to profile MPI CPU activity in Perlmutter, as the packages offered through NVIDIA Nsight are only able to profile GPU activity.

This tool allows us to extract from each simulation the total CPU time required, and its breakdown between effective time (or computation time), spinning time (or synchronization time) and overhead time (negligible for all the processor counts). Spinning time here refers to the period during the calculation when synchronizations are spinning, in other words, processors are waiting their turn to execute computations. In the following table we have the results for a benchmark simulation (fixed statistical seed) with 1,000,000 particles running on a growing number of processors in Cori, in either one or two nodes. All the units are in seconds.

Processes	Clock Time	CPU Time	Comput. Time	Sync. Time	Sync./CPU Time
16	126.272	128.349	126.376	1.295	1.0%
32	65.9231	66.64	65.909	0.73	1.1%
64	34.8969	36.780	35.286	1.494	4.1%
16x2	65.1406	66.780	66.485	0.294	0.4%
32x2	35.92	39.770	38.451	1.319	3.3%
64x2	19.8147	21.270	20.143	1.126	5.2%

First, this set of results show that we were also able to achieve strong linear scaling in Cori using our code, as we can see that in both one and two nodes, doubling the number of processors roughly corresponded to reducing in half the clock time. One interesting feature is that the communication scheme implemented is robust even when the processors are spread across two nodes, as the execution time for the code in 32 processors in a single node versus 16 processors in each of two nodes lead to very comparable results. As communication of processors across nodes is more computationally expensive, this means that our code most likely was able to implement a local communication scheme, that is, most of the data transfer is happening across closely located processors. If that were not the case, in a spread communication scheme, there would be significant data transfer across nodes, which is more computationally taxing, and efficiency of 16x2 processors for instance would be significantly lower when compared to the efficiency of 32 processors in a single node.

These results also confirm how robust our implementation has been as it is scaled across multiple processors. The synchronization time remains low in absolute and relative terms even for widely distributed schemes across 128 processors. This is consistent with better than linear scaling observed in Figure 2. In all of the cases, the synchronization time is majority composed of 'MPI Busy Wait Time', which is related to idle time for MPI processors as they wait to receive all the scheduled communication before proceeding with the calculations.

Ultimately, this time remained low across all parallelization schemes, which elucidates the robustness of our implementation.

3 Conclusion

In this analysis we were able to implement a robust MPI parallelization scheme for a 2d molecular dynamics simulation. We optimized both the spatial partitioning as well as the communication scheme between processors, and achieve both strong and weak scaling close to ideal behavior. We were also able to show that even when spread across two nodes, the desired scaling behavior is recovered, indicating minimum communication among nodes. In order to do this, we extended the code from homework 2-1 to incorporate having multiple processors. So, the core data structure used contains a grid of bins that each represent a processor and each of these bins is further divided into smaller bins to allow for faster computations. We also attempted to implement a row layout where each row represents a processor but were unable to pass the correctness tests with this implementation. With the box layout implementation, we were able to demonstrate ideal strong and weak scaling for MPI ranks from 1 to 128. To implement communications between processors, we attempted to use Gather, Alltoall, and individual send and receive. Although Gather is the least efficient out of the three, we were unable to implement the other two. However, we still got good results using Gather, as discussed in earlier sections. So, one possible extension of this work is implementing communication using Alltoall as that would cut down on unnecessary communication messages.