CS C267 Homework 2.2 Report

Maya Lemmon-Kishi, Zach Van Hyfte, and Minan Wu

Homework Group 63

1 Overview

In this report, we discuss the strategies we used to parallelize a C/C++ particle simulation program for a distributed-memory environment using MPI (Message Passing Interface). In Section 2, we discuss the design choices we made in our implementation of the parallel particle simulation. When run on two of Cori's Knights Landing nodes, using a total of 136 cores, our MPI implementation completes a $1,500,000$–particle simulation in 11.3 seconds. When run on only one Knights Landing node (using a total of 68 cores), our implementation completes the same simulation in 18.6 seconds. In Section 3, we analyze the performance of this algorithm.

2 Parallel Algorithm Implementation with MPI

2.1 Design Inheritance

As the task of particle movement simulation remains the same as last time, we adapt and reuse parts of the ideas implemented in the OpenMP program as they are proven to provide high performance. This includes `Particle Binning`, `Block Size Increase`, `Applying Symmetric Forces`, `Four Neighbors Looping` and `Rebin as Needed`. In this report we will focus on the new MPI implementation but also introduce how these ideas are adapted to the new program.

2.2 Simulation Space Grid and Processor Grid

2.2.1 2-D Implementation

- **Nested Space Division**

  Unlike the OpenMP program, the MPI program consists of multiple processes running asynchronously without shared memory. So we modified our binning strategy to be nested. The first layer assigns each process a panel covered certain area of the entire space in a 2D format. With each panel, there is a second 2D layer of tiled blocks. At the initialization of the simulation, we first assign particles to corresponding panel and then to the corresponding blocks. After the initialization, the simulation space is managed by the affinity between processes and particles.

  To reduce the communication surface area and spin up as many processes as possible, we allocated a panel grid of `row * col` size to `row * col` number of processes. We kept `abs(row - col)` small to keep the grid close to a square. We implemented this feature by finding the closest integer factors to the square root of the number of processors `num_procs`. In the cases that `num_procs` is not a square number, the algorithm will find two integers that multiplied to be `num_procs`, until one of the integer to be `1` and the another to be `num_procs`. This implementation is safe for all different numbers of processes as it supports rectangular panel dimension and rectangular block shape. In the later experimentation, we observed that for prime `num_procs`, this factorization will force the panel division to be 1-D. While this undermines the performance, it is safe to execute. As a design choice we prefer to make the code generalize well on all conditions.

- **Minimum Dimension Threshold**

  During experimentation, we observed that when simulating small number of particles on large number of processes, each process can end up having too few blocks as well as too few particles. This can undermine the performance as the overhead of communication dominates the total run time. So we set the minimum panel dimension to be 3 blocks by 3 blocks. If this condition is not met, the initialization algorithm will keep one more process idle and redo the factorization, until the minimum panel dimension is met.

2.2.2 1-D Implementation

For corner cases that did not work well with our 2-D grid decomposition scheme, we used a 1-D decomposition scheme, which divided the square simulation space into —num_procs— rows. Each of these rows spans the entire width of the simulation, and consists of `std::ceil(num_procs / l)` rows of the particle blocks into which the simulation space was divided in our previous OpenMP implementation (where l is equal to the width/height of the simulation space in particle blocks). Each processor "owns" the particles within the row whose index corresponds to its rank — it alone is responsible for calculating the forces applied to those particles at each simulation step, and for moving those particles to new blocks if necessary. This two-level decomposition of the simulation space into particle blocks within processor rows means that, as for our serial implementation from Homework 2.1, each processor does only $O(n)$ work at each timestep (where n is the total number of particles within that processor's row).

2.3 Processor Communication

We assume that particles will at most move into neighboring panels. We checked this assumption by printing out movement of particles to determine how far particles moved both in terms of blocks and panels. From this testing, we saw the largest movement of a particle was into a neighboring block and confirmed movement of most 1 panel. While more rigorous testing may show further travel across blocks, in the vast majority of cases particles do not even move out of its current block. This makes sense when dt is small, as at each step of the simulation a particle will not travel far. Since we enforce the minimum number of blocks per panel as 3, this is sufficient to show our assumption is reasonable and safe.

2.3.1 2-D Implementation

- **Point-to-Point Non-Blocking Message Passing**

  As the `Collectives` and/or `Blocking` message passing implementations cost extra time for synchronization compared to `Point-to-Point` and/or `Non-Blocking` implementation, we chose methods that hypothetically would provide higher performance. In our implementation, we directly pair up the communication sources and destinations. For each pair of communications, we use `MPI_Isend` and `MPI_Irecv` to achieve `Non-Blocking` communication. We also dedicated independent `MPI_Request` and `MPI_Staus` for each pair of communications. Each communication channel is managed independently and the message length can be retrieved independently for each data reading.

- **Ghost Region Communication**

  As each process needs to read in particle information form neighboring process to compute forces of its own particles located at the edge of panels, we implemented communication of ghost regions between neighboring processors. The processor sending this information do not allocate extra buffer memory, but instead end a pointer pointing to each specific block at its edges. The receiving processor allocates buffer memory for the ghost region of each neighboring panel as the processor doesn't own these ghost particle. Once the particle information is received, they are stored in blocks based on particles' spacial locality. By this implementation, each process will send and receive `2*panel_row + 2*panel_col + 4` blocks of particles, representing four edges and four corners of that process, shown in fig 1. This round of communication is done at the beginning of each simulation step, as only after a process receives all information from all of its ghost regions can the process begin to compute forces of its own particles.

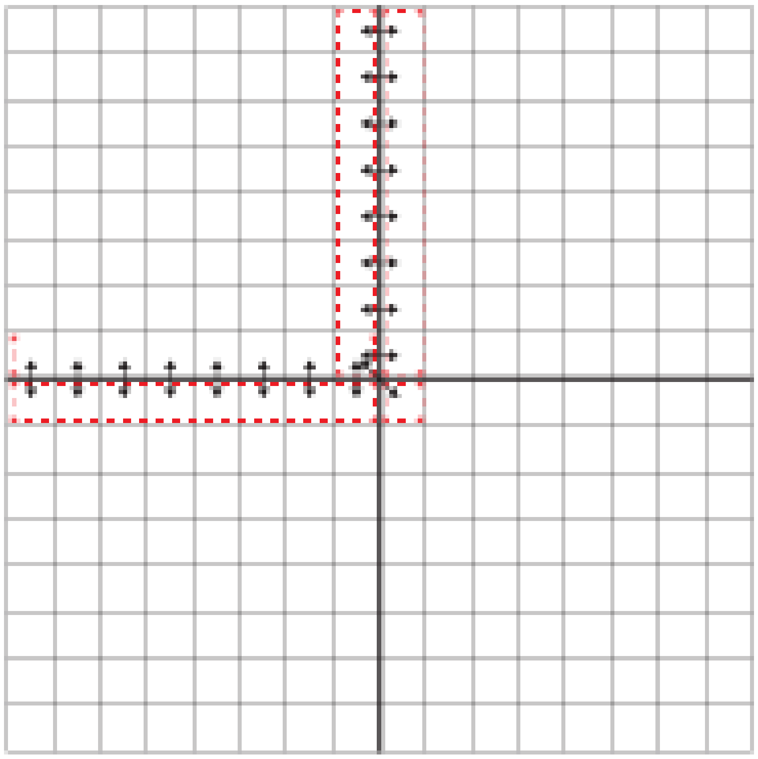

Figure 1: Ghost region communication, each block of particles is directly sent to target block

We used `MPI_Isend` and `MPI_Irecv` as communication methods and used `MPI_Wait` to wait on all pairs of communications as synchronization.

- **Particles Send-Recv Communication**

  After computing `apply_force` and `move`, some particles will move from one panel to its neighbor. We re-paneled these particles by erasing them from one process's memory and sent its information them to the new panel. As a particle can potentially have a variety of block targets when crossing panels, we used a panel as the basic unit of communication unlike the ghost region communication that used a block as the basic unit of communication (see fig 1 and fig 2). For this implementation of communication, we only allocated 8 send buffers and 8 receive buffers for each process to represent the 8 different directions of communication. This round of communication is executed after we compute `move` and determine which particles cross panels. This communication is received and synchronized after we execute local rebinning. This allows us to interleave communication time with computation time to hide communication latency and maximize throughput. After receiving all particles we then compute the target block of panel-crossing particles and bin them in their target block. The receive and binning is the last procedure of each step of in the simulation.

2.3.2 1-D Implementation

In the 1-D implementation, each processor's region shares a border with, at most, two other regions corresponding to two other processors. Since the width of the particle blocks in this implementation is `4 * cutoff` (just as in our OpenMP implementation, we determined, after testing a number of block widths, that this was the optimal width for our 1-D MPI implementation), all of the particles that could possibly be exerting force on particles in a neighboring processor row are located within the row of particle blocks directly bordering that region.

Thus, at each time step, each processor makes at most two calls to `MPI_Isend` — one to send copies of all of the particles in the topmost row of particle blocks in its owned region (”ghost particles”) to its

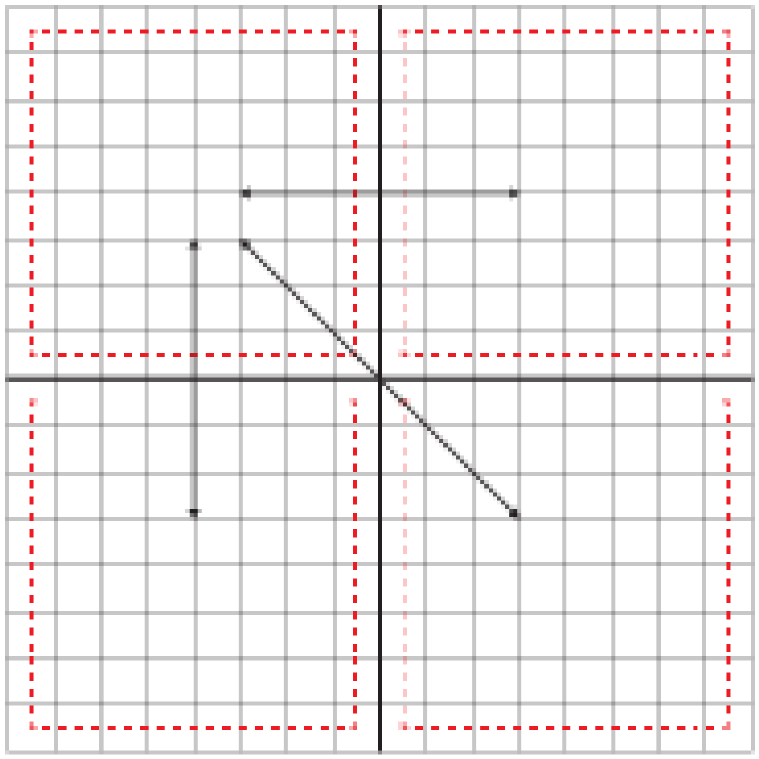

Figure 2: Particles communication, crossing-panels particles of one process are packed to send to target process

neighbor above, and one to send copies of all of the particles in the bottommost row of particle blocks in its owned region to its neighbor below. Also included within these messages are any particles that have moved out of the processor's owned region and into that neighbor's region. Each processor also calls `MPI_Irecv` at each time step to retrieve updated ghost particles from both its top and bottom neighbor processors (if they exist).

2.4 `gather_for_save` Implementation

Our implementation of the `gather_for_save` function works as follows:

1. Each process iterates through all of the blocks that it "owns" and sums up the total number of particles that it currently "owns".
2. Rank 0 collects all of the other processes' individual particle counts into an array `particle_counts`, using the `MPI_Gather` function.
3. Rank 0 then allocates an array of size `num_parts`, and calculates where each process' particles will fall within that array if each process' particles are stored contiguously, and process' contiguous sections of the array are ordered by their rank (i.e. Rank 0 calculates the `displs` parameter of the forthcoming `MPI_Gatherv` call).
4. Each process allocates a buffer `sent_particles`, large enough to hold copies of all of the particles that it owns, and then copies all of the particles that it owns into the buffer.
5. Rank 0 collects, from each process, copies of the particles that that process owns it owns particles from across all of the processes, consolidating them all into an `received_particles` array. This is done using the `MPI_Gatherv` function, called with the particle counts and displacements that were collected/calculated earlier.

6. Rank 0 iterates through all of the particles in `received_particles`, copying each particle `p` to `parts[p.id - 1]`. (This $O(N)$ procedure is faster than, say, accumulating all of the particles directly into the `parts` array and then sorting it with an $O(N\log(N))$ sort procedure).

3 Performance Analysis

Our 1-D implementation worked fairly well. Using 136 processors, it completes a $1,500,000$-particle simulation in 22.1 seconds; using 68 processors, it completes that same simulation in 32.7 seconds. However, our 2-D implementation proved to be significantly faster, completing that same $1,500,000$–particle simulation in around half the time: 11.3 seconds when using 136 processors, and 18.6 seconds when using 68 processors.

3.1 Serial Slowdown

When run on just a single core, our MPI implementation completes a $1,500,000$–particle simulation in is 973.8 seconds, and completes a $1,000$–particle simulation in 0.46 seconds. The serial slowdown of our algorithm is therefore 1.05 (Fig 3), indicating its $O(n)$ time complexity. We can see that the runtime for several other particle counts (in orange) fall on this line.

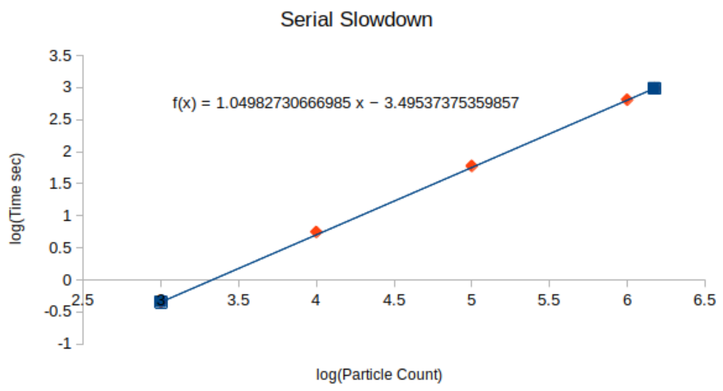

Figure 3: Serial slowdown of MPI implementation showing O(n) as the slope is 1.05

3.2 Weak Scaling

The running time of our algorithm on a $10,000$-particle simulation using 1 core is 5.6 seconds, while the running time of our algorithm on a $136\times$ larger $1,360,000$-particle simulation using 136 processors is 10.3 seconds. Its weak scaling efficiency for 68 processors is thus equal to $\frac{5.6}{10.3} = \ \sim 54.4\%$. The algorithm's weak scaling efficiency is visualized in Figure 4. While the slope of the curve is not entirely flat, the figure shows that, for the most part, there is good weak scaling at intermediate numbers of particles and intermediate numbers of processors.

3.3 Strong Scaling

When run on two of Cori's Knights Landing nodes, using a total of 136 processors, the running time of our MPI implementation on a simulation containing $1,500,000$ particles (with a random seed) is 11.3, representing a $86.3\times$ speedup over the serial algorithm.

The running time of our algorithm on a $1,500,000$-particle simulation using 1 core is 973.8 seconds, while the running time for that same simulation using 136 processors is 11.3 seconds. The algorithm is $86.3\times$ faster when using $136\times$ as many processors, so its strong scaling efficiency is therefore equal to $\frac{86.3}{136} = \ \sim 63.4\%$. The algorithm's strong scaling efficiency is visualized in Figure 5, which shows a rather linear relationship between the number of processors used and the algorithm's total runtime.

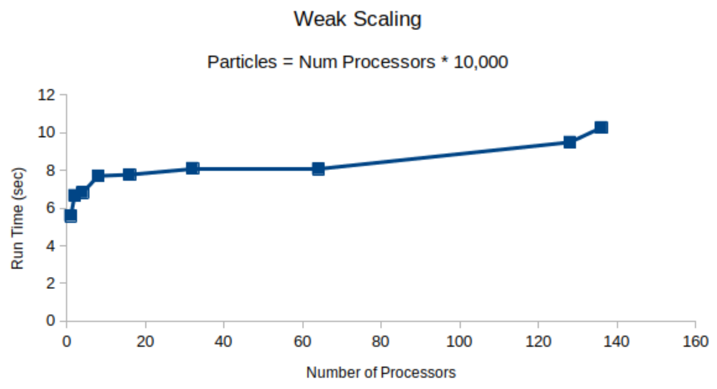

Figure 4: Weak scaling of our algorithm with particle count = number of processors * 10,000.

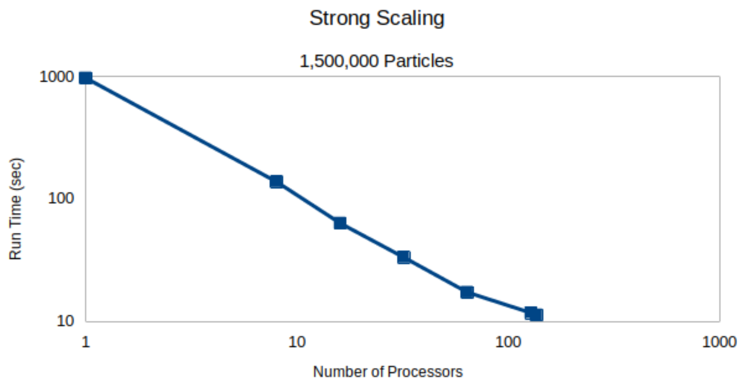

Figure 5: Visualization of strong scaling of our algorithm over various number of processors at 1,500,000 particles. The axis are in log scale

3.4 Where Did the Time Go?

Given that our algorithm's weak and strong scaling efficiencies are both far from 100%, there appears to be plenty of room to improve the parallelization strategies used in our MPI implementation. Using The University of Oregon's TAU Performance System, we profiled our implementation as it ran a $10,000$–particle simulation using increasingly large numbers of processors. Figure 6 shows how the percentage of the algorithm's total runtime consumed by calls to `MPI_Wait`, as well as the percentage consumed by communication (in our case, calls to `MPI_Isend`, `MPI_Irecv`, `MPI_Gather`, and `MPI_Gatherv`). Note that the distinction between communication and synchronization may be somewhat blurry for our algorithm, as it uses non-blocking communication; the actual sending and receiving of messages may or not happen within calls to `MPI_Wait`.

For this fixed-size simulation, as the number of processors is increased, the percent of the total runtime spent in calls to `MPI_Wait`, etc. increases — when 68 cores are used, almost 80% of the algorithm's runtime is spent in these MPI communication functions. In addition, though, as the number of processors is increased, the proportion of time spent in calls to `MPI_Isend` and `MPI_Irecv` stays constant and even starts trending downward. This also makes sense — as the number of processors increases, the size of the portion of the grid "owned" by each processor decreases, which means that:

1. Each processor has fewer particles that it needs to calculate the force upon, so computation time decreases and communication accounts for a larger portion of the total runtime

2. The number of particles that need to be sent to neighboring regions decreases, so the amount of work that needs to be done in calls to `MPI_Isend` and `MPI_Irecv` (copying data to MPI's internal buffers, etc.) decreases.

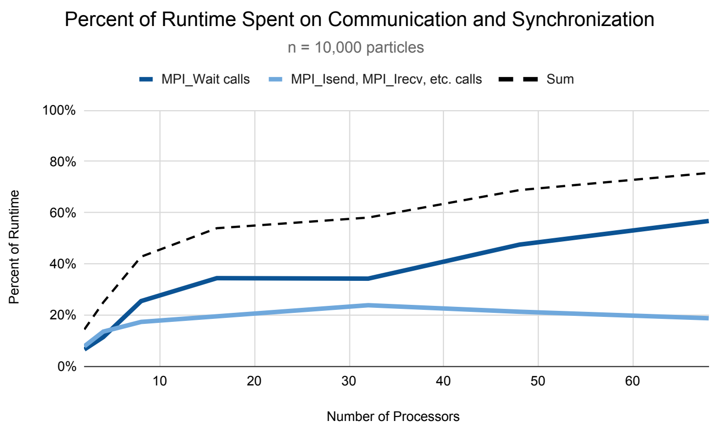

Figure 6: Percentage of runtime dedicated to communication and synchronization as the number of processors is increased

4 Team Member Contributions

The members of our team contributed equally to this report, to discussions, and debugging of our MPI implementation of the particle simulation. The individual contributions of each team member are as follows:

- **Maya Lemmon-Kishi** implemented the dynamic allocation of processors and the `gather_for_save` function.
- **Zach Van Hyfte** implemented and optimized the 1-D algorithm, and worked on an simple early version of the 2-D algorithm.
- **Minan Wu** implemented the point-to-point non-blocking 2-D algorithm.