MPI-Based Particle Simulation

Introduction

In this assignment a 2D particle simulation was implemented in parallel using MPI (Message Passing Interface) a distributed memory message passing API. The simulation was parallelized by grinding up the simulation space and assigning contiguous grid chunks to the different processors. Using 2 compute nodes and 64 tasks per node a particle simulation of six million particles was completed in roughly 73 seconds.

Team

- Justin Shao <justin.shao@berkeley.edu> - Implemented the simulate_one_step function, implemented and tested 1-d and 2-d domain sub-division.
- Ishna Barwey <ibarwey@berkeley.edu> - Plots for data visualization, performance experiments, simulate_one_step function assistance, debugging, code refactoring.
- Xavier Linn <xperrylinn@berkeley.edu> - Implemented init_simulation and gather_and_save.

Implementation

Ranks and Ghost Regions

Firstly, we designed a data structure `'grid_cell_t'` to represent a global single grid cell in our simulation. This structure represents an area of dimensions `cutoff * cutoff`. It contains the rank of the owning processor, the index of the cell, a vector of particles in the cell to calculate forces later, a set called `'is_ghost_cell_to'` containing the ranks for which the grid is a ghost cell, and a vector of the neighboring cells, including ghost region cells. This design choice allows for easy access to neighboring cells, which is used routinely for calculating inter-cell forces. By assigning pointers to neighboring cells during initialization, we reduce the overhead of searching for neighboring cells during each step of the simulation.

In initialization, we assign each grid cell to the rank responsible for its region, opting to assign nearby grids to the same rank in chunks to minimize inter-rank communication as much as possible. We divide out ***num_procs*** amount of “chunks”, each containing roughly a similar amount of grid cells, and assign one chunk to each rank. We chose to identify ghost cells by checking each grid cells’ neighbors’ ranks. After assigning a rank to each grid cell object, we also assign their neighbor grids to the static vector `'rank_ghost_grid_cells'` if such neighbor cells are assigned to a different rank than the rank of the running process. In addition, the rank of such neighbor cells is added to the grid cell’s set attribute `'is_ghost_cell_to'`.

Optimizations and Design Choices

One design choice we experimented with is to store the grid cell's neighbors in each direction as a separate attribute of the datatype (i.e. neighbor_left, neighbor_up, etc). However, we realized that this would induce 8 additional null checks and instruction branching every time we tried to discover the neighbors of a grid - which occurs once for each particle at each time step. For this reason, we opted for saving the neighbor grid cell as a vector attribute of each grid cell, and initialize it when we initialize the simulation.

Another design choice we implemented and experimented with was the domain subdivision scheme. Our earliest implementation divides the domain by rows, and assigns each processor a stripe of contiguous rows. While 1-D subdivision's performance is decent with a couple of processors, issues arise when we increase the number of processors. The border-to-area ratio is high, which induces large amounts of communication between processors. The existence of ghost cells further amplifies this issue, leading us to at times observe worse performance with more processors than with fewer processors due to communication costs.

Communication

The communication process involves sending particles from one rank to another based on the grid cell locations after each simulation step. This exchange ensures that each rank has an up-to-date representation of the particle distribution that is within the relevant sub-domains of the simulation field. Specifically, this includes the area covered by the "chunk" assigned to the rank, as well as the chunk's bordering grid cells that consist of its ghost zone.

Before communication occurs, we first need to apply forces and move the particles within the grid cells of the rank, considering each grid cell's neighbors. In this process, we make sure to tally up the particle IDs that need to be communicated to another rank. Communication needs to happen when the particle either moves into a grid cell assigned to another rank, or moves into a grid cell that is a ghost cell to some rank (i.e. borders a chunk assigned to some rank).

Our communication implementation utilized the following MPI directives: MPI_Alloc_mem, MPI_Alltoall, MPI_Isend, MPI_Irecv, MPI_Waitall. Initially, we dynamically allocate memory with MPI_Alloc_mem for the buffers used to store particles being sent to other ranks. A synchronization point is established at the blocking MPI_Alltoall collective call, which exchanges information about the number of particles to be sent between each pair of ranks. After receiving the particle counts to be received from each rank, we can allocate memory space for our receive buffers using MPI_Alloc_mem, and then proceed to exchange particle data using non-blocking communication with MPI_Isend and MPI_Irecv.

Notably, before the send and receive, we resize the send and receive buffers to fit the amount of particles being sent and received, respectively. When we proceed, we ensure that we only call MPI_Isend and MPI_Irecv for instances where the number of particles communicated is greater than 0. We tally the total sends and receives in this step, then initialize the MPI_request vectors and MPI_status vectors, which is required when we employ the MPI_Waitall call later to wait for the proper completion of all send and receive requests before proceeding.

After all send and receive operations are completed, the received particles are integrated into the current rank's local simulation state. Each received particle's information is updated in the local copy of the particle data structure (aka **parts**), and it is placed into the appropriate grid cell based on its updated position.

Finally, memory allocated for tracking particle counts is freed with MPI_Free_mem to ensure efficient memory management throughout the simulation.

Consideration on Only Sending to Neighbors

We have considered the possibility of only communicating with our immediate processor neighbors, but have decided to not implement it for the following reasons. When we only communicate with our immediate neighbors, we assume that the particles can only move up to 1 processor's region away. We have a hard time convincing ourselves that this holds generally - consider a situation where a group of particles is tightly clumped at a time step **t,** then the particle at the corner of the clump will move away extremely fast, and its position at step **t+1** is not guaranteed to be "within one processor away" under any non-trivial domain subdivision. The only strict upper bound we came up with was the borders of the entire field, which is less than helpful. As such, this change would increase performance at the cost of generality.

We do argue that the overhead of our program is not very significant when the number of processors is not extremely high, compared to solutions that implemented this approximation. The only extra calculation we are incorporating is the singular MPI_Alltoall call that helps each rank confirm the number of particles to be sent around, which are represented as integer values. The MPI_Isend and MPI_Irecv calls following this are only made when we are sure that there are more than 0 particles sent, so there are no differences in performance at this stage (in other words, any difference in this stage comes at the cost of correctness).

Performance

Below is a plot in log-log scale of our code's performance. The serial plot displays the performance on 1 processor, and the parallel line represents performance on 64 processors. We can see that as the number of particles increases, the runtime for the parallel simulation increases at a slower rate than the serial simulation. This indicates improved efficiency in parallel processing, likely due to the grid subdivision technique which can optimize the workload distribution among processors. The second plot in log-log scale displays the performance of our code with the row division implementation, which shows a steeper trend for the parallel

performance. In both cases, the serial performance shows a steeper slope, indicating that its runtime increases more rapidly with an increase in particles. We can see the benefits of a parallel implementation as the particle count increases.

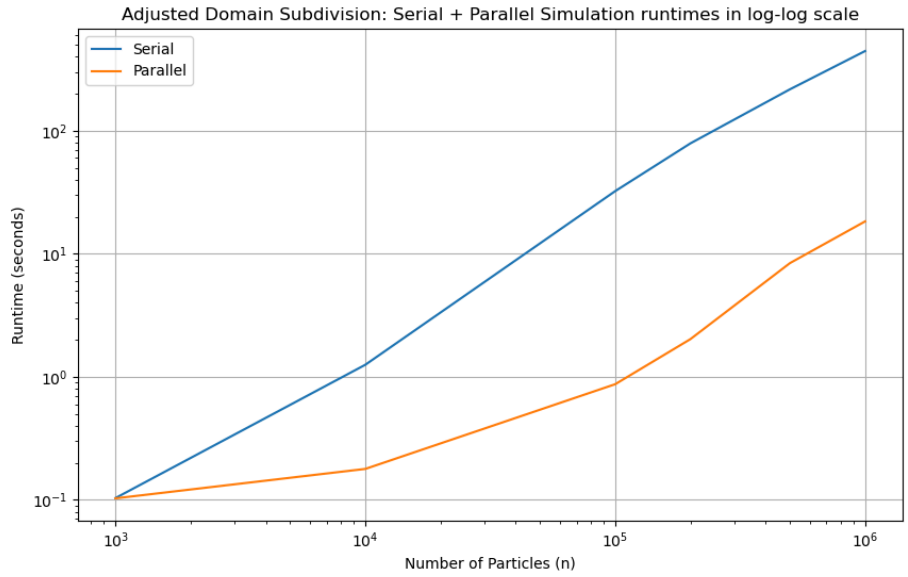


Fig.1: Final Implementation Performance in log-log Scale with Grid Rank Division

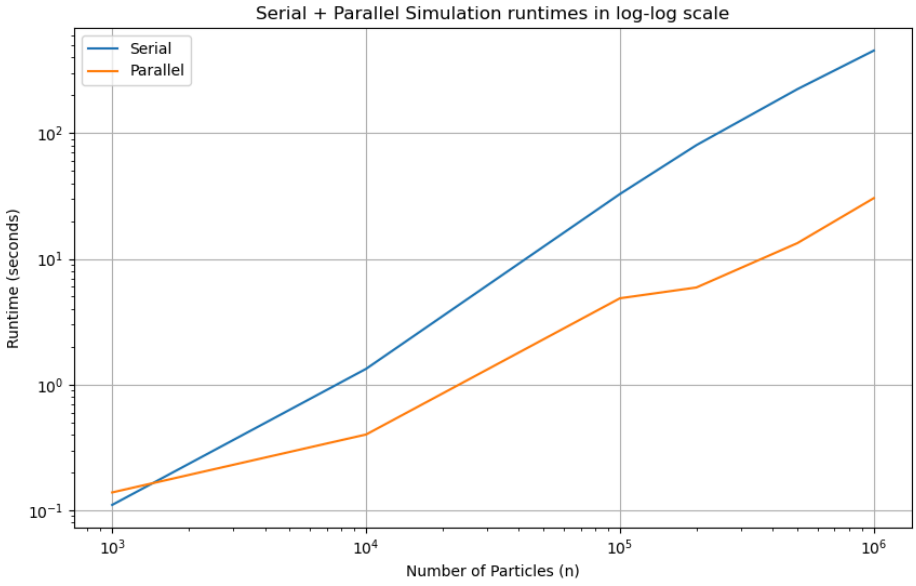


Fig.2: Performance in log-log Scale with Row Rank Division

We also evaluate the strong and weak scaling of our implementations. Below are plots of the scaling with grid division in Fig. 3, and row division in Fig. 4. We can see that the row division displays an unwanted behavior when employing 64 processors where there is a significant

slowdown compared to 32 processors. This is likely due to the fact that while adding more processors initially speeds up the computation, there is a point beyond which additional processors do not contribute to performance improvements as efficiently. This observation led us to attempt a grid rank division, which displays a significant improvement in simulation time at 64 processors to be nearly equal to performance at 32 processors for the same particle count.

The linear increase on the weak scaling plot suggests that the system is not maintaining a constant time as is the ideal scenario for weak scaling. However, the increase in time is relatively gradual and linear, indicating that the system is scaling somewhat effectively, but with noticeable overhead as the problem size grows.

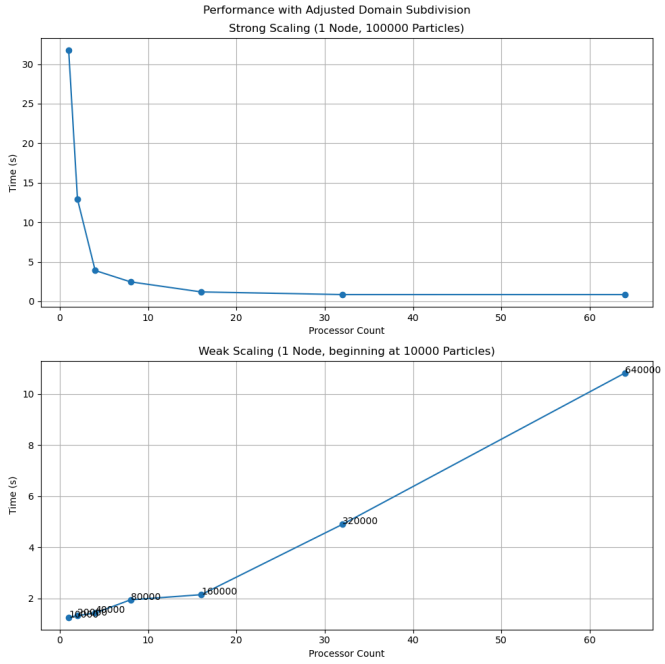


Fig. 3: Strong and Weak Scaling for Final Implementation

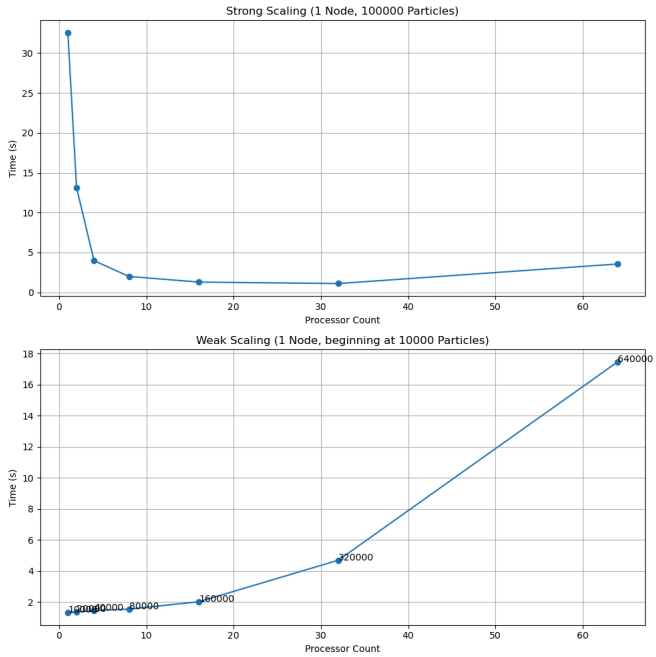


Fig. 4: Strong and Weak Scaling for Row Rank Division

Implementation of the grid division yields results of 92.3 seconds and 76.9 seconds for 6M particles, 64 tasks on 1 and 2 nodes, respectively. Compared to the original row division implementation of 112.7 seconds and 107.4 seconds for 6M particles, 64 tasks on 1 and 2 nodes, respectively.

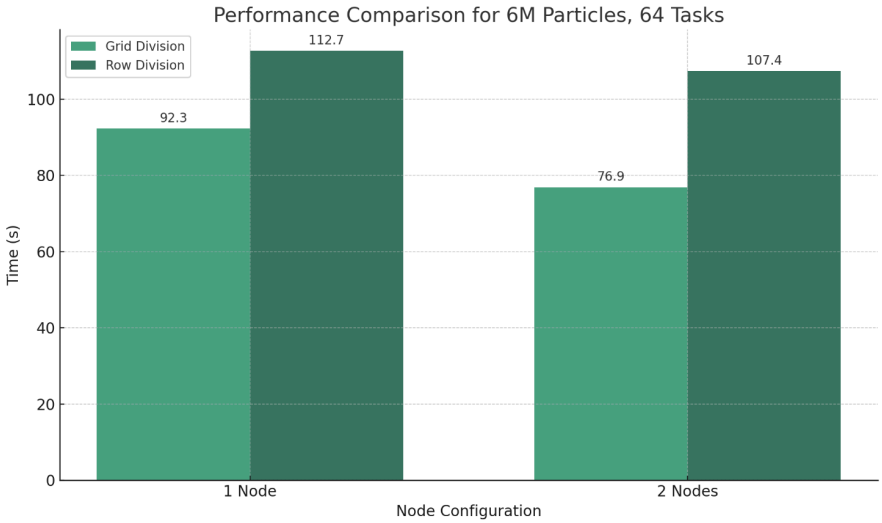


Fig. 5: Performance on 6M Particles

Breakdown

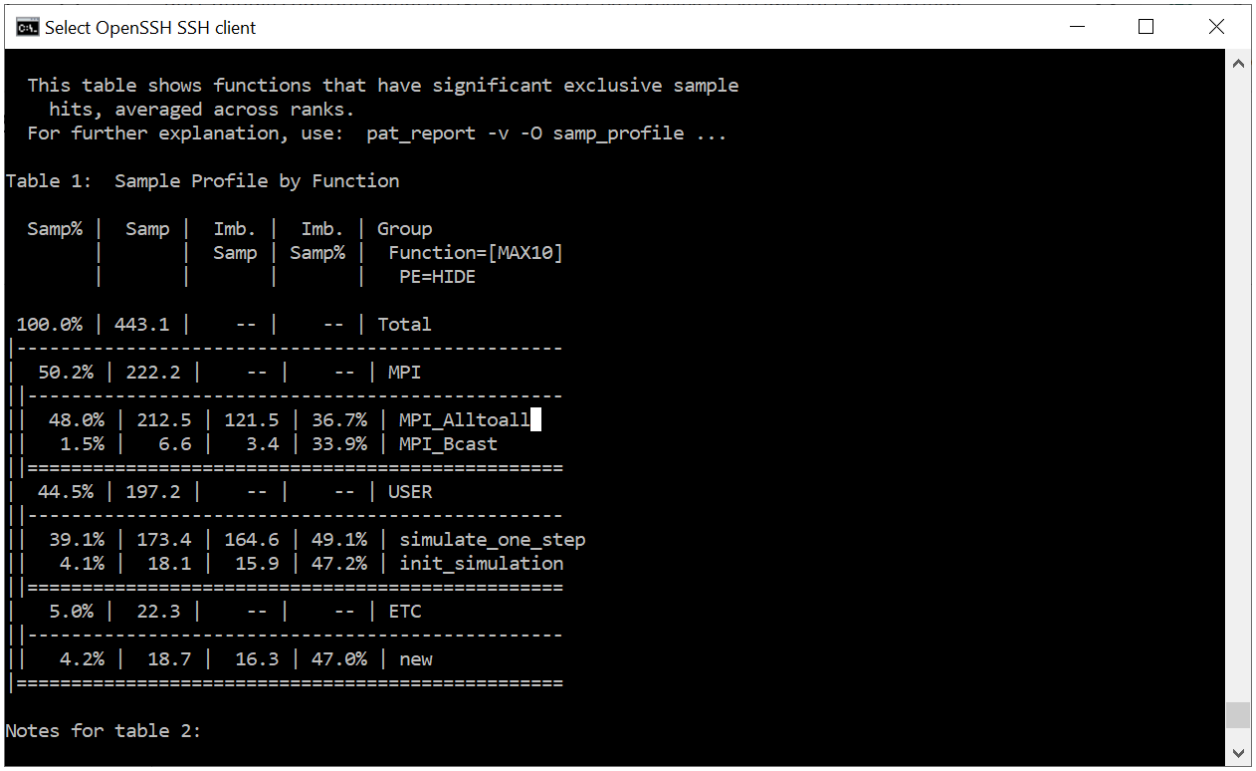
In parallel simulations using MPI, computation time tends to decrease with the number of processors as calculations for updating particle forces and positions are distributed among multiple processes and executed in parallel. However, the overall runtime of the simulation is affected by the communication and synchronization overheads inherent in parallel processing.

As the number of processors increases, the time required for communication and synchronization between processes also increases. This is primarily due to two sources: first, increasing process counts will lead to more fine-grained domain subdivision, which means that it is more likely for each particle to cross processor borders after each move. Second, before sending particle data across processors, we need to use a MPI_Alltoall directive to communicate the amount of particles that will be sent. The cost of this MPI_Alltoall call scales with the number of processes, **p**. In addition, the increased communication overhead can lead to longer overall runtime by introducing potential network congestion.

Synchronization time also increases with the number of processors, as more time is required to ensure that all processes are synchronized before proceeding to the next step of the simulation. This synchronization overhead becomes more significant as the number of processors grows, as evidenced by our strong scaling results. For example, we observe an increase in runtime when the number of processors increases from 32 to 64.

By analyzing the profiling results of running our solution with a high number of processors, we observe that the time cost of MPI_Alltoall becomes increasingly proportionally significant.

Profiling results with: node = 2, rank per node = 64, n = 200,000 (Process runtime 4.38 s)



Profiling results with: node = 1, rank per node = 64, n = 200,000 (Process runtime 3.20 s)