MPI-Based Particle Simulation

Tuesday February, 25th

We will start at Berkeley time

CS267 Spring 2025

Distributed Memory

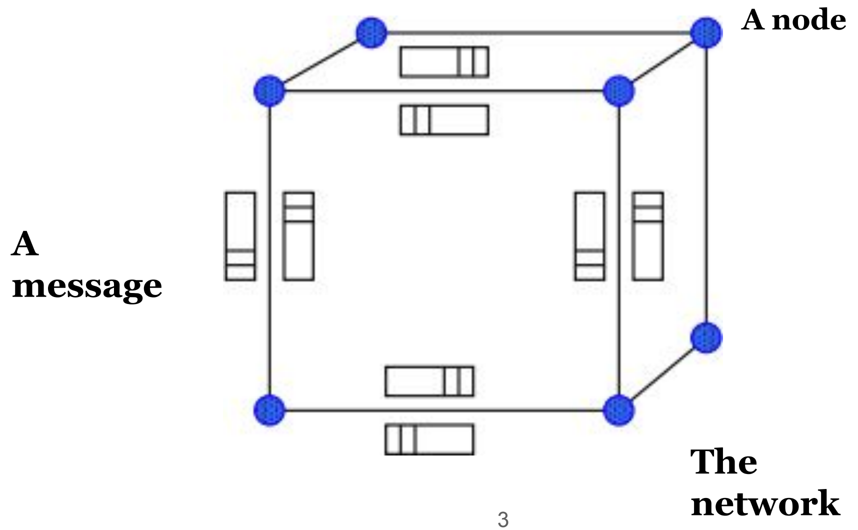
In **HW2.1**, we learned how to parallelize a particle simulation within a single node, however, **scientific problems often have orders of magnitude beyond the capabilities of a single node**.

In **HW2.2**, we want to learn **how to exploit the capabilities of multiple compute nodes** so that we can handle much larger problem sizes.

Distributed Memory

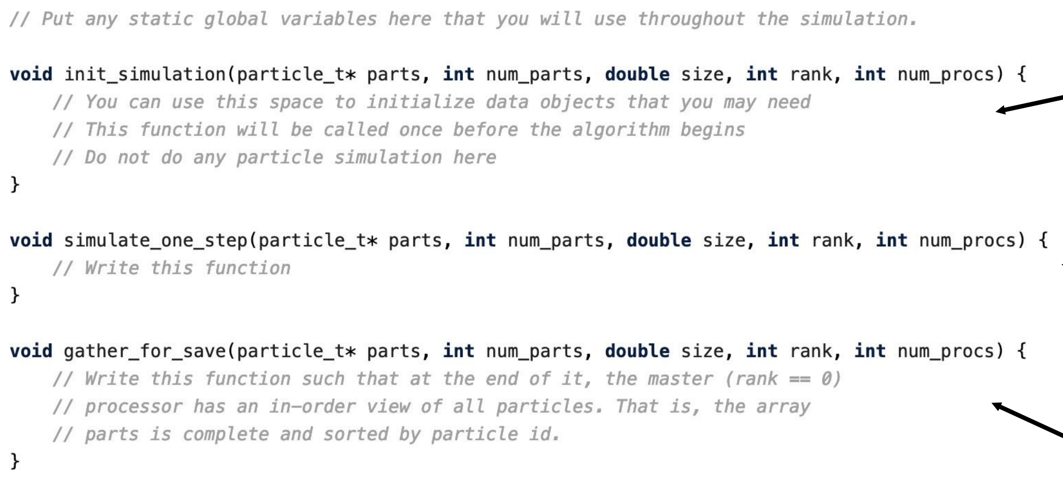
OpenMP: Threads communicate by **accessing shared memory**

MPI (Message Passing Interface): Processes communicate by a **message passing protocol**



Code Walkthrough

Starter Code



```
// Put any static global variables here that you will use throughout the simulation.

void init_simulation(particle_t* parts, int num_parts, double size, int rank, int num_procs) {
    // You can use this space to initialize data objects that you may need
    // This function will be called once before the algorithm begins
    // Do not do any particle simulation here
}

void simulate_one_step(particle_t* parts, int num_parts, double size, int rank, int num_procs) {
    // Write this function
}

void gather_for_save(particle_t* parts, int num_parts, double size, int rank, int num_procs) {
    // Write this function such that at the end of it, the master (rank == 0)
    // processor has an in-order view of all particles. That is, the array
    // parts is complete and sorted by particle id.
}
```

The input is a copy of the entire set of the particles. Each process fetches its local set in this function.

1. Each process runs simulation for its local set using the O(n) serial algorithm.
2. Redistribute the outgoing particles

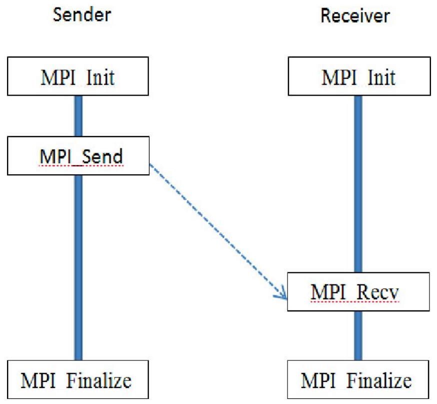
Rank 0 gathers data of all particles

Suggestions

- Copy over apply_force() and move() functions from HW2.1 into your mpi.cpp file.
- Start by implementing 1D partitioning. If you can get that working/scaling, you can try adjusting your code to use a 2D processor grid to improve communication complexity.
- MPI programming is quite different from openmp programming. Every processor is executing a separate MPI process with its own address space. This can be confusing at first because all processors are executing code from the same file. Whenever processors need access to data from other processors, you are responsible for writing the code that explicitly sends data between them.
- The particle_t struct now includes an “id” field which gives the global index of the particle. You should use this to implement the gather_to_save() function.

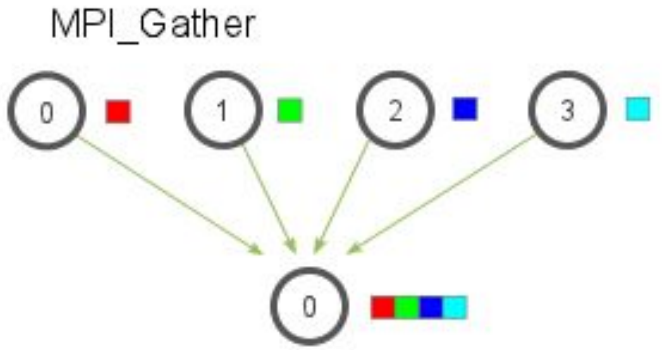
MPI Collectives and Operations

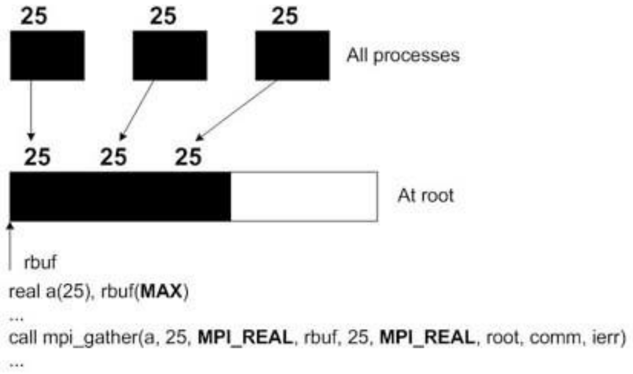
MPI_Send/Recv begins a send/receive



MPI Collectives and Operations

MPI_Gather / MPI_Gatherv gathers buffers from all processes in a communicator

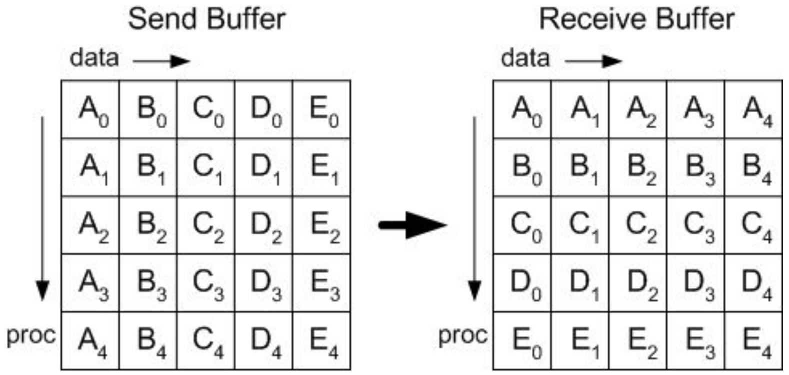




```
real a(25), rbuf(MAX)
...
call mpi_gather(a, 25, MPI_REAL, rbuf, 25, MPI_REAL, root, comm, ierr)
...
```

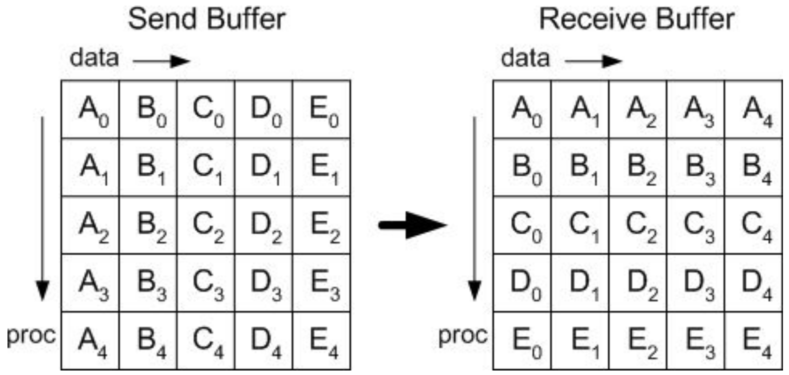
MPI Collectives and Operations

MPI_Alltoall sends data from all to all worker processes


Send Buffer
data
proc
A_0 B_0 C_0 D_0 E_0
A_1 B_1 C_1 D_1 E_1
A_2 B_2 C_2 D_2 E_2
A_3 B_3 C_3 D_3 E_3
A_4 B_4 C_4 D_4 E_4
Receive Buffer
data
proc
A_0 A_1 A_2 A_3 A_4
B_0 B_1 B_2 B_3 B_4
C_0 C_1 C_2 C_3 C_4
D_0 D_1 D_2 D_3 D_4
E_0 E_1 E_2 E_3 E_4

MPI Collectives and Operations

MPI_Alltoallv sends data from all to all worker processes — **Each process may send a different amount of data**

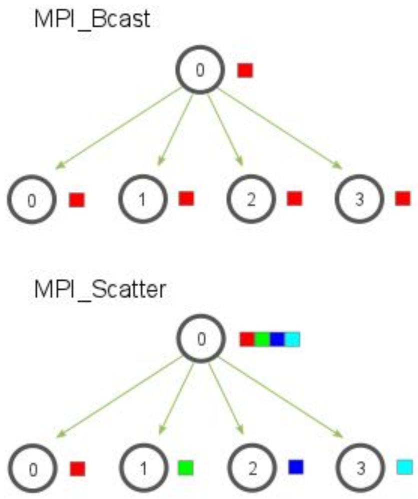


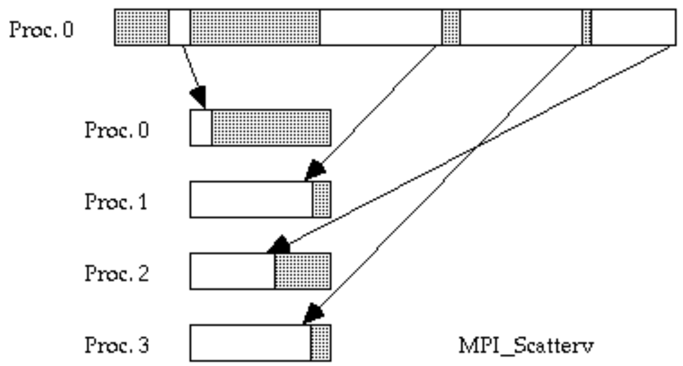
A_0	B_0	C_0	D_0	E_0
A_1	B_1	C_1	D_1	E_1
A_2	B_2	C_2	D_2	E_2
A_3	B_3	C_3	D_3	E_3
A_4	B_4	C_4	D_4	E_4

A_0	A_1	A_2	A_3	A_4
B_0	B_1	B_2	B_3	B_4
C_0	C_1	C_2	C_3	C_4
D_0	D_1	D_2	D_3	D_4
E_0	E_1	E_2	E_3	E_4

MPI Collectives and Operations

MPI_Scatter / MPI_Scatterv scatters a buffer in parts to all processes in a communicator (**MPI_Bcast** sends the same piece of data)

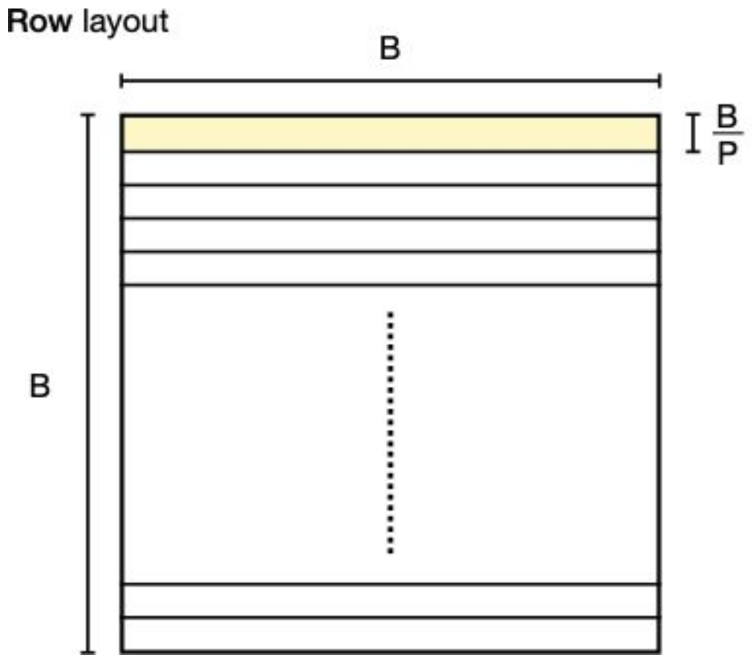


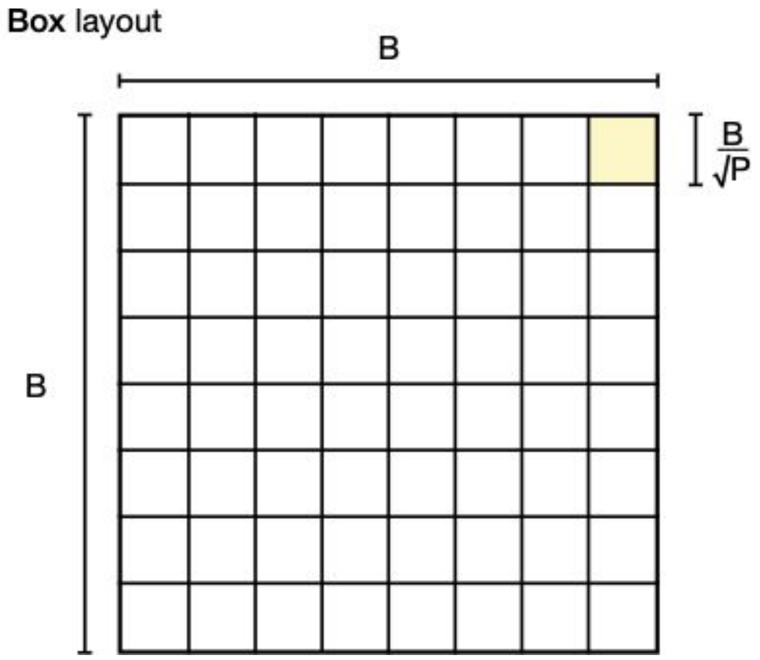


More code examples

- MPI Tutorial: https://mpitutorial.com/tutorials/

Row Layout (1D) and Box Layout (2D)



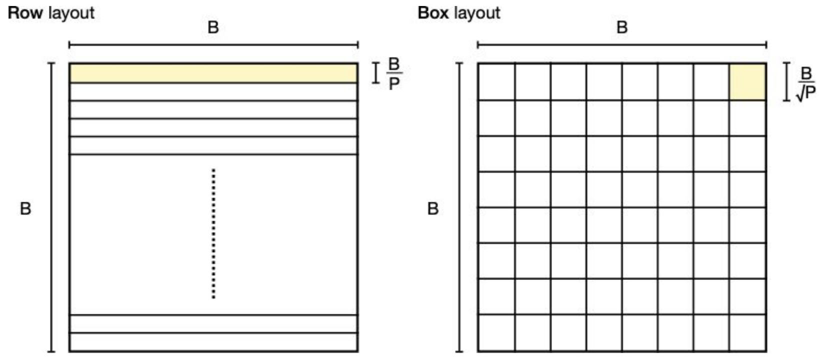


Row Layout (1D) and Box Layout (2D)

A **1D decomposition is typically more straightforward to implement** since it's easier to accommodate cases where the number of processes isn't a square number or other corner cases

However, if we consider an all-to-all communication across processes we can mathematically show that a **1D decomposition will communicate more data** then a 2D decomposition for this application

The assumptions are that the **communication is proportional to the length of the border** with neighboring processors and that **computation is proportional to the area of the owned region**



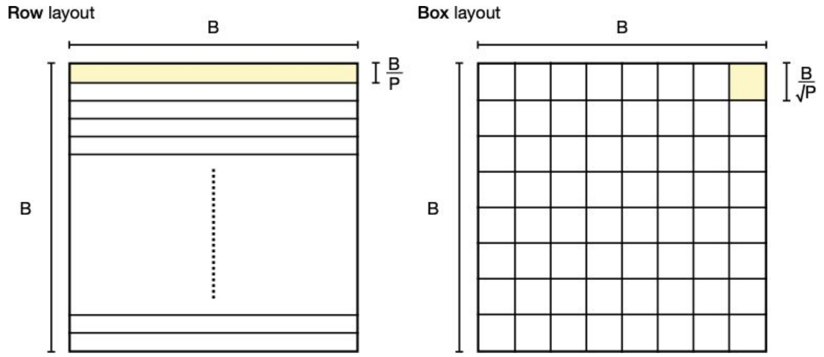
Row Layout (1D) and Box Layout (2D)

Hence, we want to **maximize the area a processor owns** and **minimize the processor borders**

In a 1D decomposition, each processor has an area of $\mathbf{B^2/p}$ where **B** is the box dimension and **p** is the number of processors while in a 2D decomposition each processor has the same area of $\mathbf{B^2/p}$, but less communication

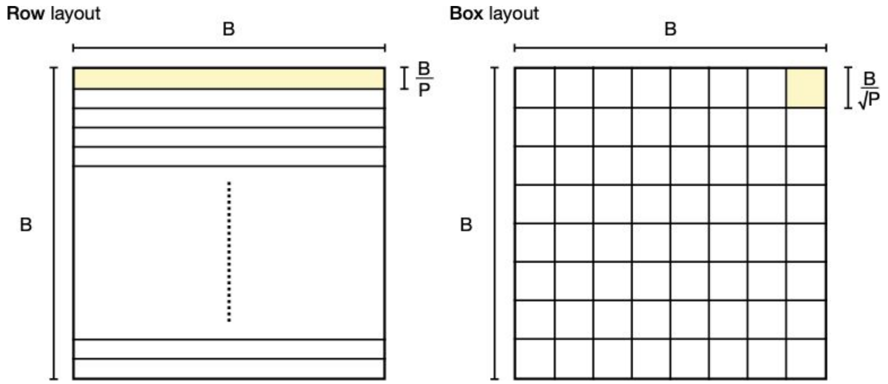
The worst case for border length for a processor in the box layout is $\mathbf{4B/p^{1/2}}$ versus **2B** in the row layout

The 2D decomposition **doesn't have to be a square**, it can be rectangular



Particle Redistribution

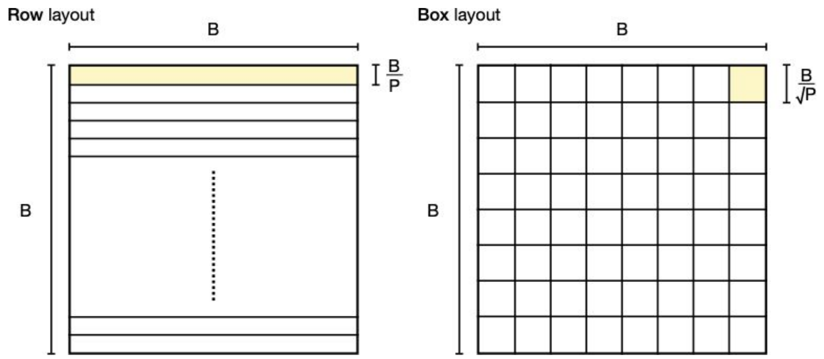
In both layouts, the particle redistribution step can be implemented using **MPI_Alltoallv**, **MPI_alltoall** or **MPI_Send/MPI_Recv**



Particle Redistribution

Based on your specific implementation, you can **probably** assume that particles cannot move “more than one processor away.”

If this is true, you could be able to achieve higher performance only communicating across neighbor processors instead of communicating with all of them (thus not using MPI_Alltoall/v but targeted **MPI_Send/Recv**)

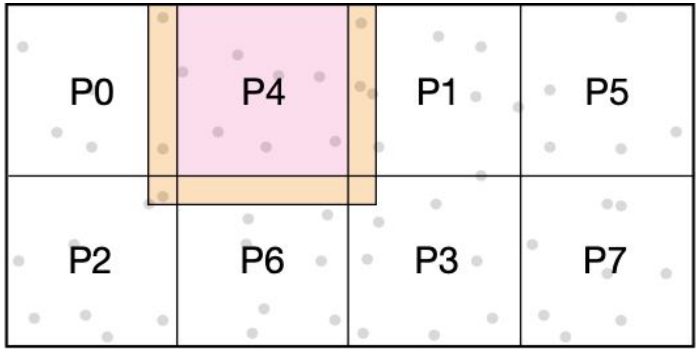


Ghost particles

A processor might not have all the needed particles and might need **particles from neighbor processors (ghost particles)** in order to compute all the forces on its own particles

Tip: To avoid communicating ghost particles across neighbor processors multiple times at the same step, you could copy ghost particles directly within each processor

This is a classic case of **trading memory over communication:** If possible, we are happy with storing some redundant data if this allows us to save up on communication

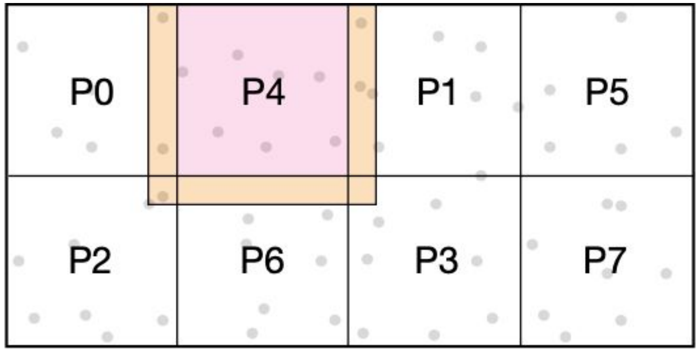


Ghost particles

A processor might not have all the needed particles and might need **particles from neighbor processors (ghost particles)** in order to compute all the forces on its own particles

Tip: To avoid communicating ghost particles across neighbor processors multiple times at the same step, you could copy ghost particles directly within each processor

Pitfall: You want to avoid communication, but **you also want to avoid redundant computation**, you don't want to compute forces over ghost particles when they're not on their owner processor



More implementation details

- Similar to HW2.1, we divide the space into bins (size is cutoff-by-cutoff) to ensure particles in each bin only need to consider other particles in its neighboring bins
- To distribute computation to MPI processors, we have to assign bins to each processor by certain layout. Assuming “row layout”, we evenly divide the space to P rows (P: #processors), and each row will cover a number of bins. (e.g., Processor i is responsible for simulating particles in those bins which belong to row_i)
- For Processor i, to simulate particles in row_i, it needs to consider some particles in some bins in row_(i-1) or row_(i+1) which we call them “ghost particles”
- In each simulation step, each processor updates particle’s next position and then MPI_Send/Receive those particles that have moved to/from neighboring bins from row_(i-1) or row_(i+1).

Grading

- Include log-log scale plots for strong and weak scaling experiments in your report.
    - Recall: **Strong scaling** experiments keep the problem size constant and vary the number of processors. **Weak scaling** experiments maintain a fixed amount of work per processor, and vary the number of processors. Ideal strong scaling speedup is P, while ideal weak scaling maintains linear runtime for each processor count.
- Include a discussion of your communication strategy in your report. How many communication phases did you use? Did you use point-to-point and/or collective routines? Did you take advantage of the processor-grid layout to reduce communication?
- Make sure your code scales, as this will be the primary way we grade with the autograder.

Questions?