Parallel Particles Simulation

Parallel and Distributed Computing

João Maçãs
Miguel Parece
Rafael Ribeiro

April 4, 2025

1 Introduction

This report details the parallelization approach used to implement a particle simulation system using MPI and compares it to the previous serial version. The program simulates particles moving in a 2D space under gravitational forces over each other with a particle-in-cell approach. Even though we don't consider every cell affecting every cell, the PIC approach is still computationally heavy, making it an excellent candidate for parallelization.

2 Parallelization Approach

2.1 Domain Decomposition Strategy

For our MPI implementation, we adopted a 1D row-wise decomposition strategy where the simulation grid is partitioned horizontally, with each process responsible for a contiguous block of rows:

```
rows_per_proc = ncside / num_procs;
my_row_start = rank * rows_per_proc;
my_row_end = (rank == num_procs - 1) ? grid_size : (rank + 1) * rows_per_proc;
```

We chose the row-based decomposition for several key reasons:

- **Communication Efficiency**: Each process only needs to exchange information with at most two other processes (those handling adjacent rows), minimizing communication overhead.
- **Load Balancing**: In a evenly distributed simulation our particle distribution per process is close to uniform, however in more distributed or clustered scenarios, it doesn't perform as good in distributing evenly.
- **Boundary Handling**: The horizontal wraparound is handled locally by each process, while vertical wraparound only requires communication between first and last processes.
- **Simplicity**: Row-based decomposition is more straightforward to implement than other alternatives while still achieving good performance.

2.2 Data Distribution

Our implementation uses a dynamic particle ownership model that adapts as particles move through the simulation space.

2.2.1 Initial Particle Distribution

At startup, all particles are initialized on rank 0 using the provided `init_particles()` function (with few changes to decide which particles go to each process). Then, each particle is assigned to the process responsible for its grid row:

Process 0 then distributes particles to their respective owner processes using MPI communication, keeping only its own particles.

2.2.2 Particle Ownership Model

Each particle has a `proc_owner` field identifying its owner process and a `cell_index` field for its position within the local grid. When a particle moves outside its current process's domain, it's flagged for transfer.

2.2.3 Data Exchange Mechanisms

For data exchange between processes we added 2 new custom MPI data types in the form of the structure `ParticleData` and `CellCOMData`.

After position updates, particles moving outside their process's domain are efficiently transferred to their new owner processes using the `exchange Particles()` method by checking the `proc_owner` field for each particle. For force calculations, processes exchange boundary cell information through `exchangeBoundaryCellInfo()`, which communicates center of mass data for cells around the current domain. This data is stored on each process in the form of "ghost cells", which the processes use to store the center of mass data of cells that aren't directly assigned to them.

These exchange mechanisms ensure correct simulation behavior while minimizing communication overhead in our distributed MPI implementation.

3 Implementation Details

3.1 Key Changes from OpenMP Version

Transitioning from OpenMP to MPI required significant architectural changes to our implementation. While the OpenMP version leveraged shared memory and thread synchronization, the MPI version needed a complete redesign for distributed memory environments where processes cannot directly access each other's data.

The key architectural differences include:

- **Process-Local Data**: Each MPI process maintains its own copy of particles and cells, unlike the OpenMP version where threads shared memory.
- **Domain Partitioning**: Instead of dynamic thread scheduling, we explicitly partitioned the domain and assigned rows to specific processes.
- **Additional Bookkeeping**: New fields like `proc_owner` and `id` were added to the `Particle` class to track which process owns each particle, and to maintain the identification of particles as they are sent between processes, since relying on the order of an array of particles no longer works.

3.2 Communication Strategy

Our MPI implementation uses both point-to-point and collective communication patterns based on specific requirements:

- **Point-to-Point Communication**: Used for exchanging boundary data between neighboring processes and for particle migration.

- **Collective Operations**: Used for global communications like initialization, synchronization, and aggregating collision counts.

For particle exchange, we use a combination of `MPI_Alltoall` to determine message sizes followed by targeted `MPI_Isend`/`MPI_Recv` operations:

```
MPI_Alltoall(send_counts.data(), 1, MPI_INT, recv_counts.data(), 1,
MPI_INT, MPI_COMM_WORLD);

// Non-blocking sends to all processes
for (int i = 0; i < num_procs; i++) {
    if (i != rank && send_counts[i] > 0) {
        MPI_Request req;
        MPI_Isend(particles_to_send[i].data(), send_counts[i],
                 mpi_particle_type, i, 0, MPI_COMM_WORLD, &req);
        requests.push_back(req);
    }
}
```

Boundary cell information is shared using direct point-to-point communication between neighboring processes, while the wrap around boundaries data is exchanged between the processes affected by it.

To optimize communication, we:

- Use non-blocking sends (`MPI_Isend`) when possible to overlap communication with computation
- Send only particles that have actually changed position
- Exchange only boundary information rather than complete grid data

3.3 Synchronization

We implemented synchronization points using `MPI_Barrier` between each major phase of the simulation:

- `updateCOM()`: Ensures all processes have computed their centers of mass before force calculations begin
- `updateForces()`: Guarantees all force calculations are complete before updating positions
- `updateParticlePositions()`: Ensures position updates are complete before particles are exchanged
- `exchangeParticles()`: Confirms all particle migrations are complete before cell associations are updated
- `updateCellParticles()`: Makes sure all processes have updated their cell-particle mappings before collision detection
- `checkCollisions()`: Each process increments its private local collision counter, which is then used after all the simulation steps to obtain the total collisions.

These barriers are needed to ensure correctness behavior of the system but it can lead to performance bottlenecks if the particles are not evenly distributed.

3.3.1 Synchronization Mechanisms

- **Barriers**: Ensure all processes in separate machines complete a phase before moving to the next.

  ```
  MPI_Barrier(MPI_COMM_WORLD);
  ```

- **Reduction Operations**: Used for accumulating values across processes.

  ```
  MPI_Reduce(&local_collisions, &global_collisions, 1,
  MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
  ```

- **'If' statements with rank checking**: Used so that different processes run different lines of code.

  ```
  void distributeParticles(long long n_part) {
      if (rank == 0) {
          // Separate and send particles to each process
      } else {
          // Receive particles from root
      }
  }
  ```

3.4 Load Balancing

Our MPI implementation uses static load balancing, where the domain is divided into approximately equal-sized row blocks assigned to each process.

Work distribution with MPI is done by manually sending the necessary information to each process. As mentioned previously, row-wise decomposition of the cell matrix is done, and each process calculates the **center of mass** the cells found in the rows that have been assigned to it, as well as calculating the forces for all the particles found in those same rows.

4 Performance Evaluation

4.1 Performance Metrics

We evaluated our OpenMP implementation against the serial version using three distinct test cases that mimic simulations with different and variable inputs.

- **SMALL**: A lightweight simulation with seed 50, 10,000 side length, 1300 grid size, 500,000 particles, and 10 timesteps.
- **MEDIUM**: A moderate workload with seed 1, 5000 side length, 20 grid size, 1,000,000 particles, and 10 timesteps.
- **BIG**: A heavy computation scenario with seed 3, 5000 side length, 50 grid size, 1,000,000 particles, and 300 timesteps.
- **Uneven Distribution**: A heavy computation scenario with seed -23, 5000 side length, 100 grid size, 1,000,000 particles, and 200 timesteps.

Each test on serial and openMP case was run on different thread configurations (1, 2, 4, and 8 threads) on an Intel(R) Core(TM) i7-9750H processor with 12 cores at 4.50 GHz. Performance metrics capture execution time in seconds, allowing us to calculate speedup and efficiency factors. We also analyzed how the simulation scales with increasing particle counts, grid sizes, and different distribution types (uniform and normal).

4.2 Speedup Analysis

The tables below show execution times, speedup, and efficiency for various thread counts across our three test scenarios.

Mode	Task	Threads	Execution Time (s)	Speedup
Serial	-	-	5.3	1.00
OpenMP	1	1	5.8	0.91
OpenMP	1	4	2.1	2.52
MPI	4	1	41.9	0.13
MPI	4	4	18.7	0.28
MPI	8	1	23.4	0.23
MPI	8	4	10.2	0.52
MPI	12	4	6.8	0.78
MPI	12	8	3.7	1.43

Performance metrics for the SMALL test case.

Mode	Task	Threads	Execution Time (s)	Speedup
Serial	-	-	144.1	1.00
OpenMP	1	1	152.7	0.94
OpenMP	1	4	44.5	3.24
MPI	4	1	30.8	4.68
MPI	4	4	20.1	7.17
MPI	8	1	32.1	4.49
MPI	8	4	24.7	5.83
MPI	12	4	29.9	4.82
MPI	12	8	24.5	5.88

Performance metrics for the MEDIUM test case.

Mode	Task	Threads	Execution Time (s)	Speedup
Serial	-	-	469.8	1
OpenMP	1	1	483.4	0.97
OpenMP	1	4	138.5	3.39
MPI	4	1	199.0	2.36
MPI	4	4	89.9	5.23
MPI	8	1	142.8	3.29
MPI	8	4	32.4	14.50
MPI	12	4	27.3	17.21
MPI	12	8	21.1	22.27

Performance metrics for the BIG test case.

Mode	Task	Threads	Execution Time (s)	Speedup
Serial	1	1	294.6	1.00
OpenMP	1	1	299.9	0.98
OpenMP	1	4	151.7	1.94
MPI	4	1	205.5	1.43
MPI	4	4	109.1	2.70
MPI	8	1	153.7	1.92
MPI	8	4	51.2	5.75
MPI	12	4	36.0	8.18
MPI	12	8	30.4	9.69

Performance metrics for the Uneven test case.

4.3 Results

Note: The tests were run in a cluster with not equal capable machines, this combined with the random way we distributed our tasks through the cluster lead to not so consistent results.

Our MPI implementation demonstrates significant performance improvements across various test scenarios. For small-scale simulations, MPI with 12 processes and 8 threads achieved a **1.43x speedup** compared to the serial version, while OpenMP with 4 threads delivered 2.52x. The medium-scale simulation showed more promising results, with MPI using 4 processes and 4 threads delivering an impressive **7.17x speedup**, consistently outperforming both serial and OpenMP implementations.

Most notably, large-scale simulations revealed the true potential of our approach. MPI with 12 processes and 8 threads achieved an exceptional **22.27x speedup**, outperforming OpenMP's maximum 3.39x speedup with 4 threads. For unevenly distributed particles, MPI with 12 processes and 8 threads reached **9.69x speedup**, showing that non-uniform distributions impact scaling efficiency.

Our implementation shows **effective scaling** with both increasing process count and thread count, particularly for larger problem sizes where the computation-to-communication ratio is more favorable. The **hybrid approach** of combining MPI with OpenMP consistently delivers the best performance by leveraging both distributed and shared-memory parallelism.

In conclusion, our row-wise domain decomposition strategy proved highly effective for larger simulations where computational work dominates communication overhead. While performance on smaller problems is more limited due to communication costs, the implementation provides substantial benefits for realistic simulation scenarios, confirming the effectiveness of our parallelization approach for real-world particle simulations.

5 Conclusion

This project successfully implemented an MPI-based parallelization of a particle simulation system using row-wise domain decomposition. Our approach achieved **excellent scalability** for large-scale simulations, with a **22.27x speedup** using 12 processes and 8 threads, significantly outperforming our OpenMP implementation's 3.39x maximum speedup. The transition to distributed-memory parallelism required implementing efficient data exchange mechanisms and careful synchronization through custom MPI data types. Performance benefits scaled with problem size due to more favorable computation-to-communication ratios, and our hybrid MPI-OpenMP approach consistently delivered the best results. Non-uniform particle distributions impacted scaling efficiency, as seen in the difference between the BIG (22.27x) and Uneven (9.69x) test cases. Future work could explore more sophisticated load balancing strategies and higher-dimensional decomposition approaches for further performance improvements.