Lattice Boltzmann Simulation with MPI Distributed Memory Parallelism

Shruti Kunder (U2657388)

1. Introduction

This work summarizes the implementation and performance evaluation of distributed memory parallelism for an MPI-based D2Q9 Lattice Boltzmann simulation on the BCp4 cluster. The baseline serial, previously memory-aligned and employing OpenMP loop-level parallelism, was hindered by shared-memory system limitations—namely cache contention and limited memory bandwidth. To avoid such bottlenecks and parallelize computation across numerous nodes, an SPMD model was used utilizing MPI. The simulation space was partitioned across MPI processes, with each rank calculating a portion of the grid and exchanging information with its immediate neighbors. Main collective operations like MPI_Reduce and MPI_Ssend were used for aggregating global data. The code was built with mpiicx, selected to be Intel-compatible with BCp4's system. Compiler flags such as -Ofast and -xAVX were used to enable architecture-related optimizations and improve runtime performance.This parallelization strategy supports scalability from a single core to many nodes, enabling orders of magnitude reduction in runtime and permitting larger problem sizes to be economically simulated.

2. MPI Parallelisation and Domain Decomposition

To take advantage of distributed memory parallelism, a row-wise 1D domain decomposition strategy was applied. Each MPI process processes a consecutive block of rows from the global grid. This configuration was chosen specifically to minimize communication volume and complexity: since the Lattice Boltzmann Method (LBM) uses a 9-point stencil, each subdomain only needs to communicate with the immediate top and bottom subdomains. The AssignedRows() function distributes the rows evenly over the MPI ranks. With an unequal total number of rows (ny), the remaining elements are distributed to the last process such that complete coverage and load balance are achieved over the simulation domain. The data of its subdomain are stored by each rank in three local arrays local_cells, local_tmp_cells, and local_obstacles. These arrays utilize a Structure of Arrays (SoA) structure, flattened for each velocity direction, where there is one single contiguous memory block per velocity direction. The Structure of Arrays (SoA) flat layout was chosen over the default Array of Structures (AoS) to enhance spatial and temporal locality, reduce false sharing, and enhance cache-line boundary alignment. Memory coalescing is primarily a GPU optimization, but this layout also enhances CPU performance by utilizing caches more effectively and enabling efficient, contiguous MPI data transfer. More importantly, this memory configuration enables effective MPI communication in that data in each direction can be moved in a single contiguous operation, with advantages of reduced MPI call overhead and increased bandwidth usage.

3. Halo Exchange Communication

The stencil nature of LBM requires that each MPI rank exchange halo rows with its immediate neighbors in order to effectively stream across subdomain boundaries. This is accomplished through a symmetric

MPI_Sendrecv pattern, which does not lead to deadlocks and maintains data order consistency. The flattened SoA layout contributes further to this by enabling each velocity direction to be exchanged as a contiguous block of memory, which reduces MPI overhead and boosts throughput. This congruence between data structure and message-passing is an aspect of the simulation's overall scalability.

4. Parallel Execution Phases

4.1. Initialization

Simulation is started by master process to read configuration parameters and the obstacle map from files. These parameters are grid size, fluid parameters, and the number of iterations that are communicated through point-to-point communication (MPI_Ssend and MPI_Recv) to all MPI ranks and initialize a simulation state between processes. The y-axis is divided with a row-wise decomposition. Near-uniform distribution is ensured by the AssignedRows() routine, adding any remainder rows to the last rank. Each process maps its local subdomain with a flattened Structure of Arrays (SoA), improving spatial locality and cache-line alignment with _mm_malloc to 64-byte boundaries, vector instruction and cache-line optimized. The master slices the global grid and barriers, assigning each slice to the corresponding rank. With explicit memory alignment and contiguous MPI transfers, cache thrashing is reduced and memory-bound features of LBM are supported by maintaining data access latency and communication overhead low.

4.2. Computation

Each timestep is initiated with halo exchange using MPI_Sendrecv to maintain boundary rows identical in consecutive ranks. Deadlock-free communication and consistency of data are provided by the blocking, symmetric function. Contiguous blocks facilitate low overhead, high communication rate, and efficient transfer of velocity data in SoA structure. After exchange, the simulation proceeds via external acceleration, collision, and streaming. Inflow row acceleration is given to the last rank. The BGK collision step, which is memory-bound, reuses constants and uses structured indexing to maximize cache use. Finally, each rank computes its local velocity average, which is then reduced to a global average by MPI_Reduce.

4.3 Collation

After the time-stepping step, all processes transfer their computed velocity histories to the master using MPI_Reduce, which totals the information and averages it over non-obstacle cells. It gives the final average velocities used for performance evaluation. Grid reconstruction is performed in gather pattern: every process sends back its subdomain to the master, reconstructing the whole domain. This is succeeded by final output generation, such as Reynolds number computation and velocity and pressure field writing via write_values().

Runtime table for different problem size (Elapsed Total Time)

Problem Size	1 Procs(sec)	2 Procs(sec)	4 Procs(sec)	8 Procs(sec)	28 Procs(sec)	112 Procs(sec)
128x128	14.56	7.48	4.13	2.37	2.80	2.67
1024x1024	846	486	253	112.5	54.27	26.4

5. Scalability and Analysis

The MPI run demonstrates fair scalability with increasing process counts and problem sizes, and dramatic performance gain when transitioning from serial to distributed computation. The speedup is not proportional to the number of processes, however, especially for small problem sizes. This is mainly due to the fact that the Lattice Boltzmann Method (LBM) is memory-bound, and the memory access overhead dominates computationally intensive behavior. Thus, increasing the number of MPI processes introduces more communication overhead that cancels out the benefits of finer-grain parallelism, especially when local problem size decreases to a point where it can no longer sustain the overhead of synchronization and data exchange. For small grids, process startup and halo exchange costs outweigh, which results in decreasing returns with more ranks. As the resolution is increased, scalability becomes better, but even then, sublinear speedup is experienced due to higher communication and probable load imbalance, especially when the number of rows to be processed altogether is not evenly divisible across all processes. The results also suggest that scaling strongly is limited after a point due to insufficient computation workload per process. To reverse this, other optimizations such as overlapping communication and computation, utilizing hybrid MPI+OpenMP methodologies, and enhanced NUMA sensitivity may increase efficiency. In summary, while the implementation exploits parallelization, performance scalability emphasizes the need for computation-to-communication balance and careful domain size adjustment in relation to the amount of processes—especially for large-core-count clusters.

6. Roofline Analysis

To complement the scalability analysis, a roofline analysis was done to ascertain whether the simulation is compute-bound or memory bandwidth-bound. From inspection of the collision() function, it was estimated that each cell update takes around 122 floating-point operations (FLOPs). In parallel with this, each cell must read from and write to 9 velocity directions within both the cells and tmp_cells arrays — 18 floats total, or 72 bytes of memory traffic per cell per pass. From this, the operational intensity (OI) was computed as:

$$Operational\ Intensity(OI) = \frac{122}{72} = 1.69\text{Flop/Byte}$$

For example, using a 1024×1024 grid, 20,000 iterations, and a 13.32 seconds execution time, the bandwidth was measured to be around 113 GB/s. Multiplying it by the operational intensity gives the resultant performance in GFLOPs/s:

$$Bandwidth(GB/S) = \frac{(n_x \times n_y \times 9 \times 4 \times 2 \times iterations)}{(runtime \times 10^9)}$$

GFLOPS/s = OI x Bandwidth = 1.69 x 113 = 191 GFLOPS/s

This analysis confirms that the provided implementation is bandwidth-bound, and that memory optimization methods such as better data locality, reuse of the cache, or overlap between computation and communication may be necessary to increase the performance further.

7. Conclusion

This work presented a parallel implementation of the D2Q9 Lattice Boltzmann Method using MPI, emphasizing 1D domain decomposition, halo exchange, and structured memory layout. Although the design considerations reduced communication overhead and cache misses, scaling experiments show that performance gain is considerable only for big problem sizes. The SoA flattened layout and memory-aligned allocation improved spatial locality and supported efficient MPI communication. Optimizations in the future may include communication overlap with computation, hybrid MPI+OpenMP parallelism, and finer-grained profiling to increase understanding of hardware bottlenecks. Overall, the MPI implementation is successful in taking the simulation beyond shared-memory boundaries, but optimization decisions need to be more rigorously analyzed and supplemented with quantitative assessments. The Roofline analysis confirms that indeed the simulation is memory-bound and hence the role of memory optimization to further maximize performance.

8. References

1. *Deakin, T. (2025). COMS30053: High Performance Computing lecture notes. University of Bristol.*
2. *Prober, M. (n.d.). High Performance Computing: Optimizing a Serial Code. University of York.*
3. *OpenAI. (2023). ChatGPT (version 3.5)*
4. *Norvig, P. (n.d.). Using MPI with C. Retrieved April 2, 2025, from https://www.paulnorvig.com/guides/using-mpi-with-c.html*