Distributed memory parallelism with MPI

Xinran Wang
1924931

April 2024

1 Introduction

This report details the optimization of the Lattice Boltzmann Method (`LBM`) code by using the Message Passing Interface (`MPI`). The optimized code is running on the `BlueCrystal`, utilizing four nodes, with each node equipped with 28 cores. This part of the code builds upon the optimized serial code developed in the previous coursework. In the previous coursework, `OpenMP` was utilized following a shared memory model, which is applied on a single node allowing multiple cores on that node to share memory and exchange data through this shared memory rather than message passing. For the current coursework, a distributed memory model is employed, which is used in multiple nodes computing environments. Each computing node in this model has its own independent physical memory. Data exchange between different nodes is explicitly conducted through networks or other communication methods. `MPI` is one of the library interfaces for nodes communication. Also, `OpenMP` and `MPI` can be used together—with `MPI` managing inter-node parallel processing and OpenMP handling multi-threading within each node. In this coursework, I used only `MPI`. By implementing the `MPI` code, I achieved performance that hit the ballparks.

2 MPI Implementation

This is a Single Program, Multiple Data (`SPMD`) model where each process executes the same code but can be initialized with different data, allowing a single program to work on multiple data sets. For this coursework, my approach is to divide the entire board into several segments, with each segment handled by one process. Once all processes have completed their calculations, the results will be integrated together.

In the initialization phase, I chose to calculate the number of rows each process needs to handle. I decided to process by rows rather than columns because the source data is stored in row-major order. Processing by rows is more convenient for coding due to continuous data indices, and since the data is contiguous, the CPU can predict and preload the upcoming data into the cache. Processing by columns involves non-continuous data, meaning that each access requires skipping a certain amount of data, which does not match the physical storage pattern of arrays. This causes each data access to potentially require loading new data blocks from the main memory. With discontinuous memory access patterns, the CPU's prefetching capabilities cannot function effectively, leading to reduced cache hit rates and increased demand and latency for main memory access.

Firstly, I allocated the last three rows to the last process. The reason for this is that upon observing the acceleration function, which only accelerates the second-to-last row of the entire board, the values of the rows immediately above and below the second-to-last row depend on the accelerated value of the second-to-last row. If these three rows are not in the same process—for example, if the last row is in Process A and the second-to-last row is in Process B—and Process A executes first, it would use the pre-acceleration values from Process B, leading to

incorrect results. Therefore, by assigning the last three rows to a single process, within this process, the second-to-last row is always accelerated first, and then the results for the third-to-last and last rows are calculated.

After allocating the last three rows to the final process, I evenly distributed the remaining rows among all processes (including process 0 and the last process). When there was a remainder, my strategy was to distribute the excess rows among processes based on their rank, from low to high, as shown in this line of code:

```
int rows_per_process = basic_work
    + (ThisRank < remainder ? 1 :
    0);
```

This method distributes the workload nearly evenly. An inefficient approach would be to have a single thread handle all the remainders, which could result in an imbalance in workload and increased runtime since the total runtime depends on the last finished process. Thus, this method would lead to longer runtimes. After determining the number of rows each process would handle, I calculated the total number of elements each process had to process. Then, I allocated memory for cells and `tmp_cells`, ensuring to allocate an additional two rows of memory to store the boundary values from the neighboring processes above and below, which are necessary for halo exchange. I then initialized each process's cells. The initialization does not need to initialize each cell's halo region, as before each `timestep`, a halo exchange is performed to receive data from neighboring processes.

Initially, I had each process read the obstacle file and store the entire obstacles, but I soon realized this was unnecessary. Reading the obstacle file repeatedly added extra I/O overhead, and storing redundant obstacles consumed additional memory. To optimize this, I modified the approach so that only the main process reads the obstacle file. The main process then slices the obstacles in the same manner as before, allocating its own segment first and then distributing the remaining segments to the other processes using `MPI_Send`. Subsequently, the other processes receive their respective obstacle segments via `MPI_Recv`. This change eliminated the need for each process to store the entire set of obstacles, reducing both memory and `I/O` overhead. `Figure 1` and `Figure 2` illustrate examples of the initial sizes of cells and obstacles for one of the processes during initialization. The yellow part represents the halo region.

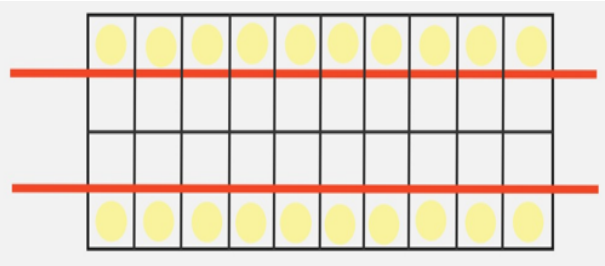

Figure 1: `cells` and `tmp_cells`

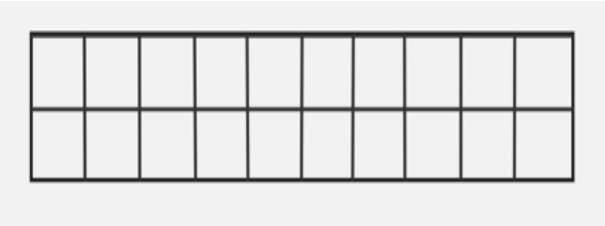

Figure 2: `obstacles`

After initializing the data for each process, I move to each timestep. At the start of each timestep, the first step is to perform a halo exchange. This involves calculating the rank values of the current process's upper and lower neighbors, and then executing two `MPI_Sendrecv` operations. The first `MPI_Sendrecv` sends data to the upper neighbor and receives data from the lower neighbor, while the second `MPI_Sendrecv` sends data to the lower neighbor and receives data from the upper neighbor. It is crucial not to write the first `MPI_Sendrecv` as both sending to and receiving from the upper neighbor, as this would cause all processes to wait for data from the upper neighbor, leading to a deadlock. Furthermore, I have customized the format for sending data as follows:

```
MPI_Datatype MPI_T_SPEED;
MPI_Type_contiguous(9, MPI_FLOAT,
    &MPI_T_SPEED);
MPI_Type_commit(&MPI_T_SPEED);
```

This is a self-defined type of 9 contiguous floating-point numbers.

Next, the `fusion_more` function is called in each timestep. In this function, it is determined that only the last process should execute the `accelerate_flow` operation. Additionally, when iterating over all elements to calculate `propagate`, `rebound`, and `collision`, it is important to pay attention to the element indices. Since I have allocated two extra rows in each cells array to store neighbors, the index range must be adjusted accordingly. In the serial code, my `fusion_more` function returns the average speed of the current `timestep`. However, now that multiple processes are handling this task, I can only return the sum of speeds for this part of the grid. After completing all `timesteps`, I then calculate the average speed for each `timestep` during data integration. The reason is that calculating the average speed per process and then combining them would follow the formula:

$$\frac{1}{|A|}\sum_{a\in A} a + \frac{1}{|B|}\sum_{b\in B} b$$

, which is not equal to

$$\frac{1}{|A|+|B|}\left(\sum_{a\in A} a + \sum_{b\in B} b\right)$$

The result we need is represented by the latter formula which means we first calculate the total sum and then determine the average.

After completing all timesteps, I need to collect the processed data from each process in the main process. Firstly, I initialized a result array in the main process, sized to accommodate the entire grid. During the data collection phase, the main process first assigns its own processed data to the result array. Then, it uses a loop to receive data from other processes. In this loop, it is necessary to recalculate the size of each process's data, as the main process cannot directly access the data size from other processes. Subsequently, other processes send their processed data to the main process. Here, I use `MPI_Send` and `MPI_Recv` instead of `MPI_Gather`. The reason is that `MPI_Gather` assumes that all processes handle the same number of elements, which does not apply in my scenario where the number of elements processed by each process are probably different. Therefore, `MPI_Gather` is not suitable.

Next, I need to calculate the average speed for each timestep. Since each process returns the total speed for that timestep, I add up the total speeds from all processes to get the total speed of the entire grid in this timestep. Then, I divide this sum by the number of non-obstacle elements to obtain the average speed of the entire grid for each timestep. For this implementation, I used `MPI_Reduce`. This function collects data from all participating processes and combines it into a single process (in this case, the main process) using a specified operation (here, I used sum).

Finally, in the main process, I perform the write operation to complete the process. Additionally, each process frees its respective resources.

3 Performance Analysis

After implementing all optimizations and using the Intel C++ Compiler (ICC) version `19.1.3.304` with the compilation command `mpiicc -std=c99 -Ofast`, the code was run on 4 nodes, each with 28 cores. The obtained runtimes for grids of different sizes were as follows: `0.99` seconds for a `128x128` grid, `1.52` seconds for a `128x256` grid, `4.53` seconds for a `256x256` grid, `9.70` seconds for a `1024x1024` grid, and `18.10` seconds for a `2048x2048` grid. It is observed that the performance hits the ballpark. In addition, according to Table 1, I made a comparison with using serial vectorization. It can be observed that using MPI to utilize multiple nodes can significantly increase the speed, espe-

cially with large datasets.

Table 1: Performance Comparison: Vectorization vs MPI(4x28 cores)

Method	Grid Size			
	128x128	128x256	256x256	1024x1024
Vectorization	7.12s	15.56s	57.78s	278.14s
MPI	0.99s	1.52s	4.53s	9.74s

	2048x2048
Vectorization	641.22s
MPI	18.21s

Note: Each data point is the average of five runs.

Figure 3 demonstrates the scalability of the program. From the graph, it can be observed that the speedup nearly shows a linear increase with the number of cores for larger datasets, such as `1024x1024` and `2048x2048`. This trend is likely due to the computational overhead significantly outweighing the inter-process communication overhead when handling large-scale data, thus making the addition of computational resources effectively enhance processing speed. In contrast, for smaller datasets (`128x128` and `256x256`), the growth in speedup gradually slows down and approaches a bottleneck as the number of cores increases. Figure 4 illustrates the parallel efficiency, which decreases with an increasing number of cores. The decline is rapid for small datasets and slower for large datasets, indicating that large datasets scale well.

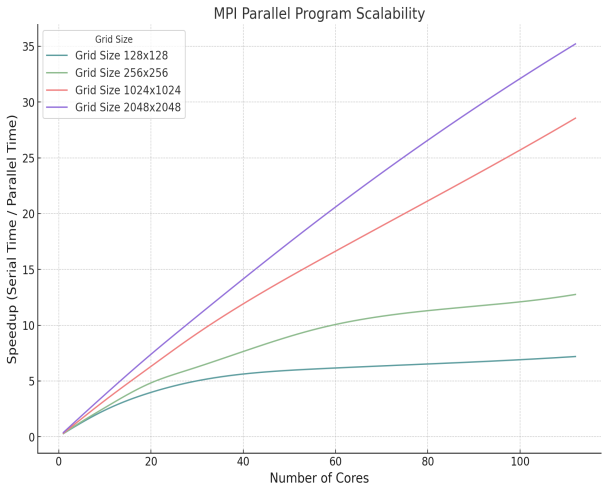

Figure 3: Scalability

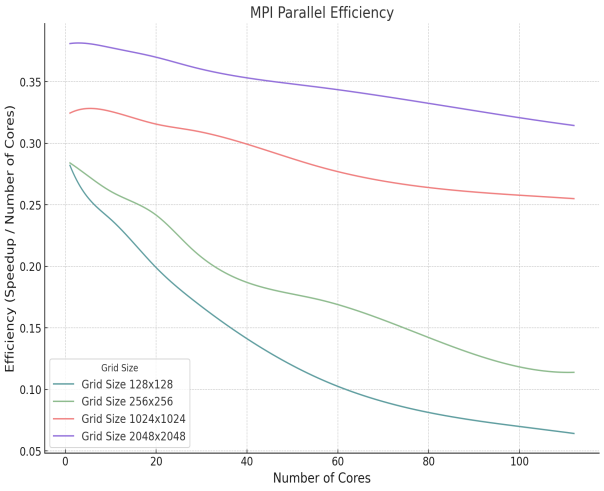

Figure 4: Efficiency

4 Future Work

During the data integration phase in the main process, the use of `MPI_Gatherv` could be explored as an alternative to `MPI_Send` and `MPI_Recv`. `MPI_Gatherv` allows the gathering of data of varying sizes. This might help in reducing overheads and simplifying the code needed for the data integration phase.

It could be beneficial to assign the main process only for data collection and integration, without assigning it processing tasks. Since the overall speed is limited by the slowest process, having the main process both execute tasks and integrate data might lead to increased time consumption.

In conclusion, in this coursework, I successfully programmed the `MPI` code to accelerate the `LBM` model, achieving good performance that hit the ballpark.