Distributed Memory Computing with MPI on 2D Lattice Boltzmann Code

Luke Zhang (az16408)
Department of Computer Science
University of Bristol
Bristol, United Kingdom

ABSTRACT

***Abstract*—This experiment focuses on parallelising the 2D Lattice Boltzmann Method (LBM) [1] with distributed memory architecture using MPI. The Single Program, Multiple Data (SPMD) paradigm is used for writing distributed memory programs. In the main computation timestep, a communication pattern known as halo exchange is deployed to exchange messages between different processes. With MPI and halo exchange running the 1024x1024 grid, the program gained a considerable 39-40x speed-up at 112 cores. It is found that this type of communication pattern scales well with larger size grid, with peak parallel efficiency at about 37.86% running with 103 cores.**

***Index Terms*—Distributed Memory, SPMD, MPI, Point to Point Communication, Halo Exchange**

I. INTRODUCTION

MPI differs from OpenMP in a way that it is a distributed memory architecture as oppose to shared memory. Because of that, the processes need to communicate with each other in order to work on solving different parts of the same problem. With this work sharing approach, we can potentially parallelise the program to as many processes and nodes as possible. It is worth noting that this might cause a lot of communication overhead if the frequency or quantity of communication is high. However, the advantage of MPI over OpenMP becomes clear when the problem size is more memory bound rather than compute bound, which is the case for the LBM [2].

II. EXPERIMENTAL DESIGN

A. Compiler and Optimisation Flags

For choosing the MPI standard, the Intel MPI Library (2019 Update 9) is used to implement the SPMD paradigm. The code is compiled using the MPI compiler wrappers `mpiicc` which in turn uses the 2020 version of Intel's `icc` compiler. As for optimisation flags, `-Ofast` is enabled to set aggressive optimisation. In addition, to optimise performance for the platform that we are running on the `-xHost` is included. Finally, the advanced vector extension `-xVAX2` is also added to vectorise the code.

B. Initialisation

First and foremost, we initialise the grid by decomposing the grid in one dimension and distributing a portion to each process (rank). In this case the grid is distributed along columns, and we decide which part of the grid belongs to each rank depending on `MPI_Comm_rank` and `MPI_Comm_size`. This way each rank has its own unique number of columns with the same number of rows across all ranks. Additionally, if the number of columns don't exactly divide the number of MPI ranks, we distribute the remainder one per rank sequentially until we have no remainder left. This is done to ensure no single rank is taking on significantly more columns than the rest of ranks, which could cause the program run slower with more variable timing when all the other ranks waiting for that one rank to finish its job (`MPI_Send` & `MPI_Recv` are blocking calls). After calculating the local grid size for each rank, the left and right ranks are also determined before allocating space for each subgrid.

In addition to allocating memory for local grid according to each rank, the halo columns are also being taken into account. Consequently, the send and receive buffers are needed to store and update

information from its neighbouring ranks in the following halo exchange step. Regarding the size of the buffer, ideally we want to pack all nine columns of speeds in a single buffer and send them with one `MPI_Send`. This is because this method is much more efficient than packing each speed in a separate buffer and perform nine `MPI_Send` individually. Therefore, the buffer is initialised in such a way that each speed can have a unique column in the buffer array. As for the obstacle grid, it is initialised as a whole grid on each rank. This is because the value of this grid stays the same throughout the main `timestep()`, which means there is no need to distribute and update it. Because of that, the value of `tot_cells` can be calculated beforehand (at the end of initialisation) since we know the obstacle grid stay the same over time. Finally, we allocate memory on the `MASTER` (in this case 0) rank for collating `cells` from all the other ranks to the `MASTER` rank, which is done after the final iteration of the main computation is complete. Although we are not initialising the whole grid on each rank, it is expected that the `MASTER` rank might take slightly more time to complete initialisation. This is due to the extra collation grid it needs to allocate memory for besides its own portion of the cells grid.

C. Halo Exchange

Shortly after the `accelerate_flow()` is executed, the leftmost and rightmost (excluding the halo region) columns of this rank can be propagated into the halo region of its neighbouring ranks. It is noteworthy that the order of ranks respect periodic boundary conditions, which means the last rank wraps around the first rank. As discussed in the initialisation step, all nine speeds are set to be packed in a single buffer array for the two edge columns of each rank (exclude halo). This enables us to update a halo column with a single send and receive. To prevent deadlock from occurring, we must make sure every `MPI_Send` has a matching `MPI_Recv`. For this reason, `MPI_Sendrecv` is used for the halo exchange.

The purpose of halo exchange is to ensure every cell in this rank (exclude halo) will have eight surrounding `cells` before going into the main computation in the `timestep()` fucntion. This is crucial for updating the new values of `cells` correctly in each iteration. To begin with, all the ranks first pack their leftmost column from `cells` (exclude halo) to a send buffer. The packing step is then followed by a `MPI_Sendrecv` call, which send the values from the send buffer to the rightmost halo column of all the ranks. After that, all the ranks make a receive call (with the same `MPI_Sendrecv`) in order to unpack from a receive buffer to their own rightmost halo column. Next, this process is repeated in the opposite direction with a second `MPI_Sendrecv` call, which means all ranks send to their right neighbours and receive from their left neighbours. By the end of this process, each rank should have the most up to date `cells`' values from its neighbouring ranks stored in this rank's halo region.

D. Timestep (Main Computation)

With SPMD, each rank runs the same program but operates on its own assigned portion of the data. After updating the halo region of each rank, all the ranks are now able to compute their new `cells`' values individually and correctly. In this part, we first specify the indices of axis-direction for each cell's speed. The north and south indices will still wrap around the local grid. However, the east and west indices will no longer need to wrap around due to the existence of the halo region. As the obstacle grid is initialised as a whole grid on each rank, each rank will only need to iterate through a portion of the obstacle grid as required. Each `cells` speed then decide what value to write to the `tmp_cells` depends on the value of obstacle, which in turn corresponds to the same position in that portion of designated `cells`. This way each portion of the `tmp_cells`' values can be computed and updated simultaneously across all ranks. From here, each rank can obtain new values of `cells` after the pointer swap is performed on each process.

E. Reduction

For computing the value of `av_vels` at the end of each `timestep()`, there are two values that we require, namely `tot_u` and `tot_cells`. In our initialisation step, we have already computed the value of `tot_cells`. As a result, this saves us from computing the same value repeatedly in the `timestep()`. Unlike OpenMP, the value of

tot_u is not a shared variable but rather a different record computed separately on each rank. For this reason, we need to use a MPI_Reduce across all the ranks to reduce all the values down to a single one before dividing tot_cells. To achieve this, all the ranks other than the MASTER rank execute a MPI_Reduce right after the main timestep() is done.

With regard to the arguments for the reduction function, the send_buffer is set to &tot_u and receive_buffer is simply NULL for all ranks other than the MASTER rank. For the reduction function arguments on the MASTER rank, a special MPI_IN_PLACE is used to indicate that the send and receive buffer are one same buffer. In addition, the receive_buffer is set to &tot_u to receive values from other ranks. With these settings in place, all ranks other than the MASTER rank will make a MPI_Send call to the MASTER rank, while the MASTER rank make a MPI_Recv call to collect the tot_cells values computed from other ranks. Then, a MPI_SUM operation is performed on the MASTER rank to sum all the tot_cells values from other ranks (including itself) and produce a single tot_cells value. After the above steps, the value of av_vels is computed via dividing summed tot_u by tot_cells. This marks the end of one reduction for this timestep before the next one begins.

F. Collation

It is often a good practice to reduce communication between ranks [3] and thus perform as much local computation as possible. For packing cells from each rank and unpacking cells to collated_cells on the MASTER rank, a similar method as the halo exchange can be used to send and receive the cells' values and minimise communication frequency. Only this time we pack the nine speeds for all the cells on that rank (exclude halo), along both columns and rows instead of a single column. In this case, we would only make one MPI_Send call on each rank other than the MASTER rank, followed by a matching MPI_Recv on the MASTER rank. Once the send and receive part is done the whole grid can be collated from different ranks starting with the MASTER rank. Because the collation function only execute once near the end of the program, it is not necessary to be concerned about communication overhead. Hence, we do not expect the elapsed time for collation to be very noticeable especially for grid with smaller size. After the collation step is done, we can proceed to calculate the reynolds number and write av_vels and final_state values to a file on the MASTER rank.

III. Results

A. Performance Comparison: MPI vs Optimised Serial

In the first column of TABLE I, it can be seen that the optimised (include vectorisation) version of the program achieves an average run-time of 6.67s for 128x128 grid, 42.45s for 256x256 grid and 218.68s for 1024x1024 grid. As the serial version is ran on only one core, large size grid cost significantly more time. The reason for that is our program become more memory bound as the grid size grows, therefore fetching data more frequently between memory and processor which in turn costs more time. Looking at the second column of the table, the speed-up resulted from integrating MPI into our program is very dramatic. It can be shown that all three inputs sizes gain drastic speedup after running the program with MPI at 112 cores across four nodes. In comparison to the optimised serial version of the program, the 128x128 grid runs 7.85x faster while the 256x256 one gains even more speed-up at about 12.67x. To take thing even further, the 1024x1024 grid manage to extend the speed-up to a momentous 34x at 112 cores. At this point, it is apparent that the MPI version do not scale the speed-up at the same rate for all three inputs. This is likely because communication between ranks takes up relatively more time for grid with smaller

TABLE I
Run-Time Comparison with mpiicc: Optimised (&Vectorised) Serial(1 core) vs MPI(112 cores)

	Compute Time (s)	
	Optimised Serial	MPI
128x128	6.67	0.85
256x256	42.45	3.35
1024x1024	218.68	6.44

size. Even though the grid with larger input size has more data to send for the halo exchange, the communication amount is relatively small compared to the amount of computation.

B. Scalability of MPI Parallelism

In the final section, we analyse the scalability of our parallelised MPI code and quantify the parallel performance for all three input sizes. The speed-up for all three inputs is calculated against their MPI version running with 1 core as baseline. According to the scaling graph in Fig 1, it can be shown that the run-time with input size of 128x128 and 256x256 both scale up rather slow compared to the 1024x1024 one. For the 128x128 grid, the line rises steadily to 4x at around 18 cores before slowly approaches 5x when the number of cores is increased to 112.

As for the scaling graph for the 256x256 grid, it can be observed that a similar trend unfolds as the one in 128x128. Although in this case the speed-up rises sharply to about 10x at around 32 cores, followed by a slowly rising long tail which eventually hits at roughly 14x. The reason both 128x128 and 256x256 grid plateau earlier than 1024x1024 is because both grid still benefit from additional memory bandwidth from more cores. Nonetheless, this is true only until they reach a certain (smaller) number of cores. As this is where the communication overhead outweighs the benefit and they start having diminishing returns by adding more cores.

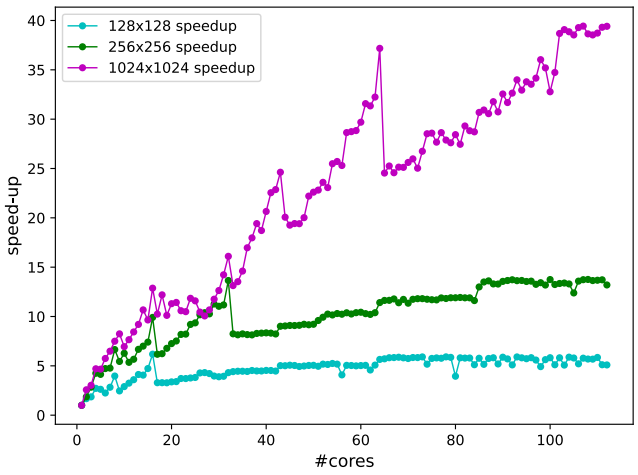

Fig. 1. Scaling with MPI: 1-112 cores

However, the trend for the 1024x1024 grid paints a much more different picture. The speed-up surges tremendously to more than 39x at around 100 cores before plateauing at this level at 112 cores. This again confirms that larger input size scales better as more cores being added. This is likely because as we increase the number of cores, the communication to computation time ratio for larger grid increases at a slower rate than smaller grid. Additionally, larger grid also benefit more from the additional bandwidth as it is even more memory bound. Finally, we can deduce the maximum parallel efficiency for this program (when the speed-up for 1024x1024 grid first hit 39x) as follows:

$$PE = \frac{speedup}{\#cores} = \frac{39}{103} = 37.86\%.$$

Therefore, the maximum achieved parallel efficiency for this program is 37.86%.

IV. Conclusion

To conclude, this experiment designed and tested a distributed memory programming method with MPI for a given LB code. In the performance comparison section, the MPI version with 1024x1024 input achieved a considerable 34x speed up than that of the optimised serial version. After analysing the scaling graph in the following section, we reached a reasonable speed-up of 5x for 128x128, 14x for 256x256 and more than 39x for 1024x1024. From there, we found that this program scales well with large size grid which hit a maximum parallel efficiency of 37.86%. This is due to fact that larger grid benefits more from the extra memory bandwidth which results from adding more cores. It is of importance to be aware of the trade-off between the amount of communication overhead and speed-up for smaller size grid.

References

[1] S. Succi, M. Sbragaglia, and S. Ubertini, "Lattice boltzmann method," *Scholarpedia*, vol. 5, no. 5, p. 9507, 2010, revision #151625.

[2] N. P. Tran, M. Lee, and S. Hong, "Performance optimization of 3d lattice boltzmann flow solver on a gpu," *Scientific Programming*, vol. 2017, 2017.

[3] Tips and best practices – introduction to parallel programming with mpi. [Online]. Available: https://pdc-support.github.io/introduction-to-mpi/13-tips-and-best-practices/index.html