Distributed memory parallelism to optimise and scale Lattice Boltzmann algorithm using MPI

Mohamed Saad Eddine El Moutaouakil (jq20457)

Abstract: In this paper we will discuss the implementation of a scalable parallel version of the Lattice Boltzmann algorithm using MPI. The starting point is a serially optimised implementation in C. The experiments are designed to be run in the Blue Crystal Supercomputer V4 with 4 nodes, each with 2 sockets yielding a total of 112 cores. For a better software/hardware compatibility, the Intel MPIICC compiler will be used with the flags *-Ofast* and *-xAVX -qopenmp*.

1. Domain Decomposition

The Lattice Boltzmann algorithm relies on 2D grids modelizing the cells and the obstacles. Then, it performs a highly parallelisable simulation iterating on each cell and obstacle. Trivially, each MPI rank -representing a core- will process a chunk from the initial grid. However, two problems arise. First, there is three different ways to distribute the grid across the ranks: by rows, by columns or by tiles as shown in figure 1.

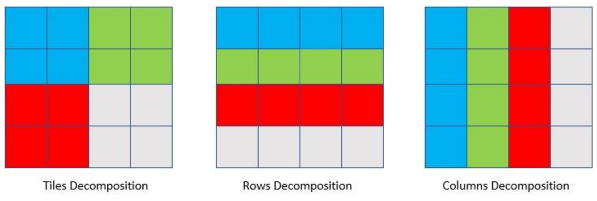

Figure 1 : Grid decomposition possibilities

Second, the computations needed in the simulation of one cell require speeds from the 8 surrounding cells. The selection must then be based on the most efficient distribution in terms of minimising the communication costs without compromising the correctness of each computation.

While the tiles decomposition can in many applications give optimal communication costs, it is the costliest choice for Lattice Boltzmann. In such choice, each rank will need to exchange information with the other 9 surrounding ranks. On the other hand, the row distribution and column distribution require only each rank to communicate with its adjacent rows and columns successively.

Since the grid is in fact stored as a one-dimensional array contiguously in the memory – with adapted indices to model a 2D grid-, the optimal choice would be a row distribution which will preserve the contiguousness of data in memory and follow the same indices pattern.

The number of rows that each rank will process depends on the size of ranks. In order to ensure a balanced load, the rows will be split evenly across the ranks. If there are any remainders, they will be distributed one-by-one to ranks starting from the master rank. Specifying a single rank to process all the remainder rows has been avoided because of its unbalanced load distribution resulting in unnecessary delays.

Consequently, in the initialisation process the memory allocation needs to be adapted to the number of rows each rank will process – augmented with two additional rows for halos that will be discussed in a later point -.

The obstacles grid however requires a slightly different approach due to the fact that each rank needs the total number of free cells to execute the collision step. I chose to assign this task to the master rank – rank 0 – where the whole obstacles grid is read, and the free cells are computed. The result is simply casted to all the other nodes through the MPI function *MPI_Bcast* and returned by the *initialise()* function to keep this value in the main function.

Just before returning from the *initialise()* function, the master rank scatters the obstacles grid to all the ranks accordingly to the number of rows each rank is processing and their offsets. For this reason, two arrays containing successively the number of rows per rank and the offset per rank have been created.

2. Halos Exchange

As mentioned above, the collision step imposes a data dependency between each row and its two adjacent ones. Thus, the boundary rows for each rank must be communicated to the ranks that need it - the previous rank and the following one-.

This is done in the main loop using two calls to the *MPI_Sendrecv()* function. As shown in figure 2, the first call propagates the halos from the bottom ranks to the upper ones, while the second call propagates them in the opposite direction.

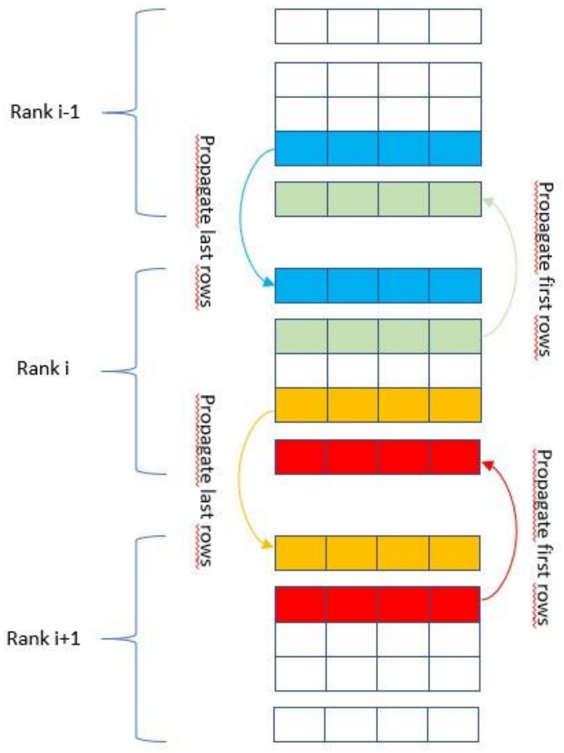

Figure 2 : Halos Exchange

Every rank stores the halos it received in the original cells' grid in the first and last row, which naturally imply a shift by one to the indices of all the reads and writes to the cells' grid.

It would have been possible to use separate *MPI_Send()* and *MPI_Recv()* functions, but it will not yield any performance difference while increasing the risk of deadlocks.

The *MPI_Sendrecv()* function remains in fact the most adequate MPI blocking communication operator thanks to its deadlock-free property.

A third and probably more efficient strategy would be the usage of non-blocking communication. Each rank would initiate a halo exchange procedure and without waiting, processes the cells that are independent of the halos - the second row, and before-last one - which will amortise the cost of communication by overlapping it with computations. This strategy was explored but not implemented with the belief that it will not significantly improve the runtimes - as recommended in the 4th lecture of MPI-.

3. Reducing averages

In the serial algorithm, each iteration of the main loop yields an average value computed at the end of each collision step. In the parallel version, every rank processes its own chunk of the grid. The primitive strategy is that all the ranks send their partial results, to one specific rank – the master rank for example – which will sum them to yield the global average value for that iteration, and store it in its array of averages.

This strategy works thanks to the distributive property of multiplication over addition. In fact, since the average is computed using the total sum of the norm of x and y velocities divided by the number of free cells – which is constant- the average can be written as the sum of partial averages as shown in the equation below.

$$\frac{\sum \|\vec{u}\|_i}{\#free\_cells} = \sum \frac{\|\vec{u}\|_i}{\#free\_cells}$$

A more elegant and optimal way to compute averages is to create a temporary array where each rank will store its partial averages every iteration. And only after the maximal number of iterations, sum the partial averages using the *MPI_Reduce()* function and place the averages obtained in the original array in rank 0.

4. Writing to the output file

The last step of the Lattice Boltzmann algorithm is to write the velocity norm for each cell to the output file. Because of the choice made of allocating to each rank -including the master- only a part of the cells' grid, the function *write_values()* needs to be adapted.

Three strategies can be implemented in order to allow a correct I/O output.

First, gathering all the velocity norms in the master rank, and use the same function *write_values()*.

Second, synchronise the writing process so that each rank writes its part of the grid completely before the next one starts.

Last, it is possible to use MPI I/O library to write in parallel to the same output file. However, since the performance desired is only defined within the scope of the main loop and don't include the writing process, I preferred for simplicity to choose the second strategy.

To do so, I created a function *synchronise_writing()* which opens the file for writing for the rank 0, and allow each rank, one by one, to write the velocity norms. *MPI_Barrier()* was used to prevent the ranks from writing simultaneously to the file.

5. Results and scaling analysis

As tables 1 shows, there is a remarkable speedup of 6.75x, 10.42x and 38.04x respectively for the sizes

128x128,256x256 and 1024x1024 when the number of cores changes from 1 to 122 cores.

	Serial optimised runtime	Runtime with 122 cores
128x128	10.26s	1.52s
256x256	56.73s	5.44s
1024x1024	307.02s	8.07s

Table 1: Runtimes for different sizes using 1 core and 122 cores.

Size of the problem influences the speedup when the number of nodes is significant. It is due to the fact that the higher the size, the easier is to mask the communication cost because the CPU spends more time in processing the cells. This becomes harder and harder when adding more computational power since the ratio of time spent on processing to communication cost decreases.

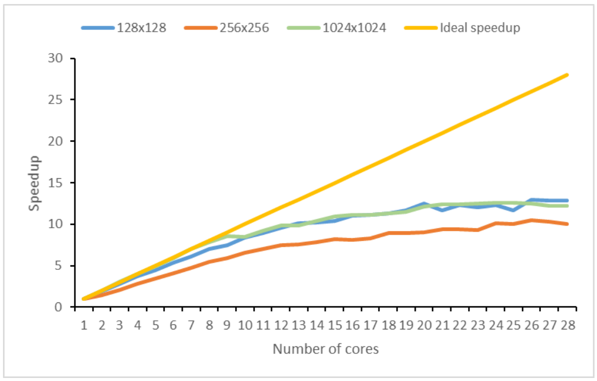

Figure 3: Performance scaling for Lattice Boltzmann algorithm in one node of Blue Crystal

Scalability is the measure of efficiency to any parallel computing task. The expected pattern of an increased speedup when adding more cores is indeed observed in figure 3. We notice however that the implemented algorithm follows the linear trend only until 8 cores. From that point it slowly increases to reach a plateau around 12x speedup.

We can also notice that when only one node is used, the speedups of 128x128 and 1024x1024 are very close. However, when 4 nodes are used the speedup of 128x128 drops to 6.75x while the speedup of 1024x1024 jumps to 38.04x.

This behaviour is the result of the inter-node communication cost which give the advantage to big sizes since the computations are significant enough to hide this cost compared to small size where the inter-node communications become the bottleneck.

Concerning the 256x256 size, we would have expected it to follow the same trend as 128x128 and 1024x1024 sizes, however we notice that it reaches slightly lower speedup rates while following the same pattern as the other sizes.

This is the result of the practical experimentation of the code in Blue Crystal's nodes. The runtimes vary significantly (up to 2.5x) depending on which node is actually executing the code. Due to the increased demand on the *veryshort* partition in Blue Crystal, each fixed size problem was benchmarked in a different node. At the time of writing, the set of nodes [compute094-097] are the fastest and most stable ones.