Optimizing The Lattice-Boltzmann method with Open MPI

Carlo Alessi
Student ID: ca16296

1. Introduction

The aim of this project was to parallelize the Lattice-Boltzmann method to run on high performance hardware, using Open MPI, which is one of the most used library implementation of the Message Passing Interface (MPI).

In section 2 I discuss further serial optimizations useful for the parallel execution. In section 3 I report the details of the performance gained by implementing different versions of the Halo exchange pattern and how to exploit the Open MPI API. In section 4 I report a wide range of experiments showing the difference between serial and parallel performance and how it is affected by workload imbalance. Section 5 concludes with a short discussion of the main results and a insight of the future directions.

2. Further serial optimizations

First, this study investigated the difference between single and double precision floating points. Since float occupy 32 bits and double occupy 64 bits, from a serial perspective, the latter gave more accuracy of the results, while the former allowed to fill the cache with twice the array cells, increasing the likelihood that a local grid fit entirely in a node cache. From a parallel point of view, it halved the total amount of bytes sent among processes decreasing the network latency. Moreover, it helped to fill the network bandwidth without reaching its plateau, which would have slow the process. This optimization gave a speed-up of 1.1.

It was found that the variable *tot_cells* was constant throughout the program execution so that there was not need to compute it each time step. The program performed the computation once at the initialization step. This change impacted more on the parallel performance rather than the serial one since it removed the needing to reduce the values at the end of each time step, like was naively done before. This gave a small speedup of 1.03 for the serial code.

Computing the square root of a number is known as a hard problem. However, this study did not put effort in optimizing it, as the profiler used (VTune) did not highlight the operation to be intensive. The reason is that the application dealt with small numbers.

The program compiled with the Intel compiler ran around three times slower than the application compiled using the GNU compiler.

3. Design of the parallel application

One approach would be to create a Master process that handles grid decomposition and recomposition and let Worker processes do the computations. However, as the aim of the project was to compute as fast as possible it was wasteful to spawn another process just for collecting the results. The design adopted in this application allowed to keep all processes busy.

The processes initialized the main grid and allocated space for their local grids. Each rank had its own local grids for cells, scratch space and obstacles each of which were initialized with respect to the main grid, the number of columns and rows assigned, and rank position. The processes sent boundary conditions to the neighbors. Once the solution completed, each rank sent its results to the Master.

The research evaluated the difference between row-wise and column-wise decomposition.

3.1 Column-wise decomposition

First, the grid was decomposed by columns. The Halo Exchange step was performed at the beginning of each time step, so that each process executed accelerate_flow(), to preserve the correctness of the program. Several execution of the application, on 1 core and on 64 cores, on different problem sizes, gave a speed-up of 15, 16, 30 and 38, respectively from the smaller grid to the bigger one.
It was clear that the application did not scale well, as the aim was to achieve a speed-up as near as possible to the number of cores used. The reason was that the Halo exchange was too intensive as each rank packed

the halo columns into buffers before sending them, and then unpacked the received data into its local grid.

Indeed, this pattern of decomposition is more suited for FORTRAN because it stores arrays in column major order. The use of the C programming language, that handles storage differently, lead to row-wise decomposition.

3.2 Row-wise decomposition

In C, as it stores arrays in row major order, it was easier to decompose grids by rows rather than columns. In principles, since rows are stored in a contiguous block of memory, the research exploited the memory spatial locality. This made the Halo exchange tidier and faster as now was performed by a single pair of MPI_Sendrecv(), avoiding the overhead of the previous for loops for packing and unpacking the columns that caused a scattered access of memory. This increased the speed-up by 39% for the biggest grid.
Once the time steps completed, the partial solutions were combined in the Master process. Each rank called the collective function MPI_Reduce() to add the average velocities of each time step. It was important to specify the length of the buffer containing the velocities to be equal to the number of time steps so that only one reduction was needed.

After that, each Worker sent its entire local grid to the Master process with a single call to MPI_Send(), matched on the other side by a call to MPI_Recv(). The total number of Send() and Recv() was equal to the number of Worker processes since the Master process just copied its own local grid into the main grid.

The optimizations discussed above for row-wise decomposition gave different speed-ups for different problem sizes. The speed-ups achieved were 23, 32, 34 and 53, respectively from the smallest grid to the biggest one. The first fact observed was that the application scaled better as the problem size grew, the reason was that for small problem sizes it was not worth it to decompose the grids because the I/O operations were more intensive than the actual computations. The second thing noticed was that the speed-ups achieved were far away from the expected results on running on 64 cores. This was because the application was still I/O-bound rather then CPU-bound due to the time spent in the Halo exchange and recomposition step.
The profiler used (VTune) uncovered that the most active functions in the application were memcpy() and MPI_Send(). Since the former is highly optimized[1], the next goal was to investigate different types of I/O trying to hide its overhead as much as possible overlapping computations and communications.

3.3 Different types of I/O

Point-to-Point communications occur in any MPI application and it is key to choose between different types of I/O.
First, the study experimented MPI_Ssend() to make the program safe and portable, but since it was a synchronous and blocking operation, processes might be idle waiting for a matching MPI_Recv().
It was found that the Open MPI implementation used (version 1.6.5) provided a system buffer, as the use of MPI_Send(), which was also blocking, but asynchronous in this case, gave a speed-up of 1.03. The research did not consider MPI_Bsend() because an asynchronous and blocking I/O was already achieved. Also, attaching and detaching the buffer would carry extra overhead, increment the code complexity and raise portability issues.

The last experiment carried out investigated whether non-blocking I/O was suited for this application. In the Halo exchange step, the MPI_Sendrecv() pair was replaced with two pairs of MPI_Isend/Irecv() so that the each rank could send and receive the data at the same time, in both directions. Previously each rank exchanged boundary conditions with just a neighbor at a time. This only lead to small speed-up of 1.01 because the call to MPI_WaitAll() was placed right after the Halo exchange step disallowing the computations to be overlapped by the communications. Indeed, the profiler uncovered that the time spent by processes on MPI_WaitAll() made up double the time spent on MPI_Isend().
Then, it was found the correct way to overlap communications and computations. Once the I/O started, the propagation step was performed only from

1 In fact after I removed memcpy() I achieved only a speed-up of 1.01.

the third row to the third last included, which were the inner local cells that were independent from the Halo exchange step. When the propagation of the independent cells completed, the program waited for the missing data to be received. After that MPI_Waitall() returned almost immediately, as the Halo rows were safe to do computation on, and then the program propagated the top most inner row and the bottom most inner row of the grid. However, the program was marginally slower than the previous version. Although the CPU time of MPI_WaitAll() made up 0% for the previous version of the program and accounted for 2.9% for the new version, meaning that the overlapping succeeded, the lack of improvement might be because of cache misses occurred when propagate() was computed on the top and bottom row, or because the overhead to control the flow of the additional loops erased the benefits. The use of the *-funroll-loops* compiler flag did not improve the speed of the program.

Regarding the recomposition step, the Master process called MPI_Irecv() for each Worker, which matched the function calling MPI_Isend(). While the data was flowing behind the scene, from one node to the Master, each rank reduced the average velocities calling MPI_Reduce(). At the end of the reduction, each rank waited for the I/O to be completed and then resumed the execution. Even tough it was expected a speed-up, because I/O was parallelized, several execution of the program showed that the program was actually slower than the previous version that used blocking I/O, resulting in a speed-up marginally less than 1. It might be either because the bandwidth reached a plateau, or the reduction step was lightweight compared to the actual grid transfer, or the implementation of non-blocking I/O through a pool of threads added delays due to context switches[2].
The final version of the program did not time the recomposition of the grid and this did not make any substantial difference.

4. Experiments

In the following sections, I give a quantitative measure of the optimizations carried out, comparing the program scaling and the elapsed time of the column-wise and row-wise division as more processes are added. Then I discuss about load imbalance when the size of the cohort does not divide the problem size.

4.1 Elapsed time

Figure 1 shows the elapsed time gained when the program executed under optimal conditions, that was when the problem size was split evenly among all processes. The chart shows that row-wise decomposition was always faster than the column-wise decomposition.

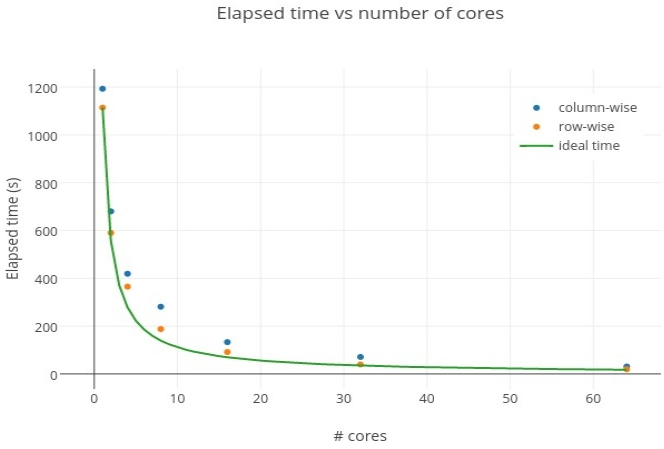

Figure 1: elapsed time for row-wise and column-wise tested on 1, 2, 4, 8, 16, 32 and 64 cores.

4.2 Scaling

Figure 2 shows the Speed-ups achieved for a evenly divided problem sharing, emphasizing the fact that not only the row-wise decomposition was faster than the column-wise decomposition, as showed in Figure 1, but it also scaled better. The blue straight line represents the ideal scaling of a general MPI application, where for twice the cores the program goes twice as fast.

2 This depends on the implementation of MPI_Isend. It could be also implemented by kernel threads.

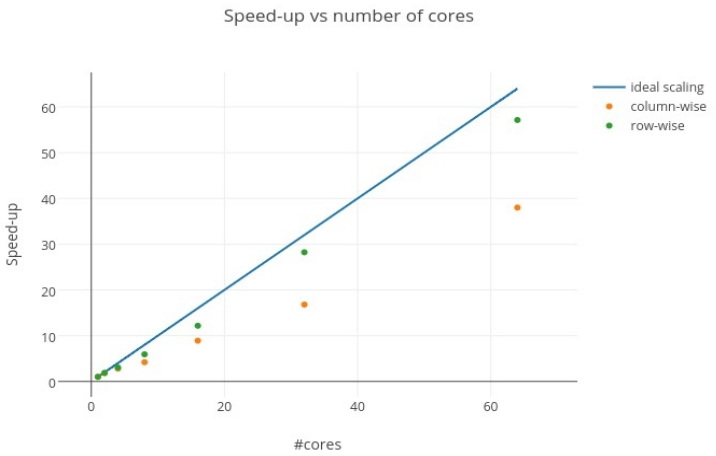

Figure 2: Speed-up for column-wise and row-wise tested on 1, 2, 4, 8, 16, 32 and 64 cores.

In the next section I put the focus just on the row-wise version of the program, but the same principles apply for any MPI application.

4.3 Workload imbalance

The program was expanded to work on any number of cores from 1 to 64. Different approaches have been investigated to split the number of rows. The chart in Figure 3 shows the performance gained from different approaches compared to the ideal scaling.

Figure 3: speed-ups achieved for row-wise decomposition from one to 64 cores highlighting the load imbalance.

One manner to address load imbalance was to assign all the remaining rows to the last rank. However, this approach led to an unfair distribution of the work among processes, since the overloaded process automatically became the bottleneck of the application, affecting the speed and the scaling of the whole program in these non-optimal scenarios, breaking the ideal strictly increasing behavior of the scaling line. In this case, the load imbalance was a function of the problem size and of the number of processes involved.

Another study approach was to equally distribute the remaining rows among the processes starting from the rank 0 to consume any leftovers. It was possible because (nrows Mod N) is a number between 0 and N– 1, where N is the size of the cohort, so that each process from rank 0 up to N – 2 received an additional row to work on, and the last process was the only one without an extra line assigned. In this case the workload imbalance was constant because the number of lines assigned differed by at most one. Indeed, according to the orange curve in Figure 3, the speed-ups were consistent when this approach was adopted.

4.4 Results

Table 1 summarizes the final results for all the test cases:

Size	Serial (s)	64 cores (s)	Speed-up
128 x 128	32.62	0.95	34.19
128 x 256	65.87	1.53	42.99
256 x 256	256.79	6	44.30
1024 x 1024	1127.20	19.68	57.27

Table 1.

5. Concluding remarks

The research uncovered that the row-wise decomposition performed better than the column-wise decomposition. The optimizations carried out did not allow the application to reach the ideal scaling because it was not found a manner to properly exploit non-blocking I/O.

Further studies are still open as the research did not cover Hybrid Programming with OpenMP and Open MPI. The idea would be to use MPI across nodes and OpenMP within nodes, that would be useful to address load balancing and to reduce memory traffic.