Distributed Memory Parallelism with MPI on the Lattice Boltzmann Simulation

Paolo Mura
rk19763

I. Introduction

In this report, I detail my attempt at using MPI to parallelise the Lattice Boltzmann fluid dynamics simulation. The original source code was a serial program written in C. In a previous assignment, I applied serial optimisations to the code, vectorisation and parallelised it using OpenMP. Ultimately, the result was a modified version of the code that could run on all 28 cores of a node on Blue Crystal phase 4, which achieved up to 94X speedup when compared to the original serial version.

As a high-level overview, this code stores two grids in memory: *cells* and *tmp_cells* (scratch space). Its main program iterates for a specified number of time steps. Within each time step, it makes a call to a function called *timestep()*, which performs the main logic of the simulation. Specifically, this involves propagate, rebound, collision, average velocities and accelerate flow steps. This is where the majority of the optimisations take place.

Since then, I refactored the code to make use of MPI parallelism rather than OpenMP. This involved splitting the grid across different ranks during initialisation, writing a halo exchange system, reductions and gathering the results into a final grid.

The results showed a massive speedup when compared to the optimised serial runtimes and surprisingly outperformed the OpenMP parallel version, even with the same resource allocation.

Throughout the report, I shall use the following abbreviations: LBM (Lattice Boltzmann), BC4 (Blue Crystal phase 4).

II. Optimisations

The general idea behind using MPI is that it allows multiple processes to each run an instance of the code. Each process can run on a separate core in parallel, as well as distributed across separate nodes of BC4. Ultimately, this optimises the code further than was done with my previous OpenMP implementation, which used a shared memory system rather than the distributed version that my MPI approach uses. The higher level of optimisation is due to being able to parallelise the grid even further than was possible when limited to just 28 cores on a single node.

A. Initialisation

MPI allows for multiple processes to each run an instance of the code in parallel. It provides functions for each process to be able to ascertain its own "rank" (id) number and the total number of processes working in the system. I set this information at the start of the program initialisation.

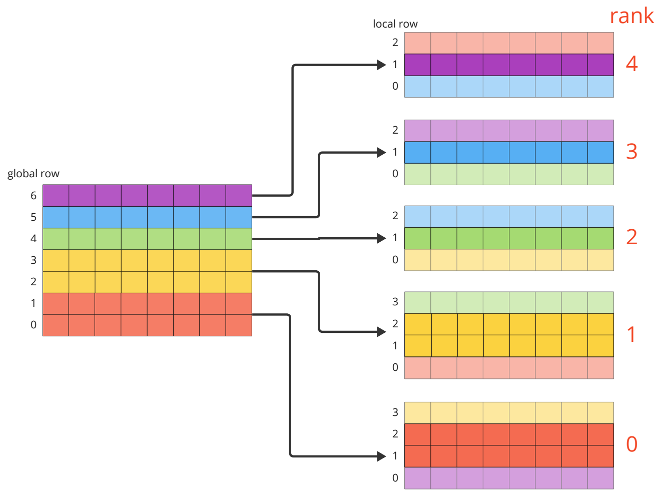

Fig. 1. Row allocation for an 8x7 grid across 5 ranks.

After this, each process calculates its allocated portion of the grid using this rank information. My allocation algorithm simply divides the number of rows of the grid by the number of processes. Each process is allocated this number of rows, with the remainder distributed as a single row per rank, starting from rank 0. On top of their own allocated rows, each process is also allocated two additional rows: the row below and the row above its designated region (the "halo" regions whose usage is explained in section II-B). This is illustrated by example in Fig. 1.

Once the assignment has been determined, each process uses *_mm_malloc* to allocate enough space in memory for its designated rows, including the halo regions, for both *cells* and *tmp_cells*. Because of this, no single process needs to store the entire grid; just the bare minimum required for them to process their own rows and to store buffers for rows communicated to them by neighbouring ranks (see section II-B). The rank 0 process alone allocates itself additional space that is enough to store the entire grid (for gathering final results as discussed in section II-D).

B. Halo Exchange

The *timestep()* function was mostly left unchanged, aside from adjusting the range for the loops since they each now

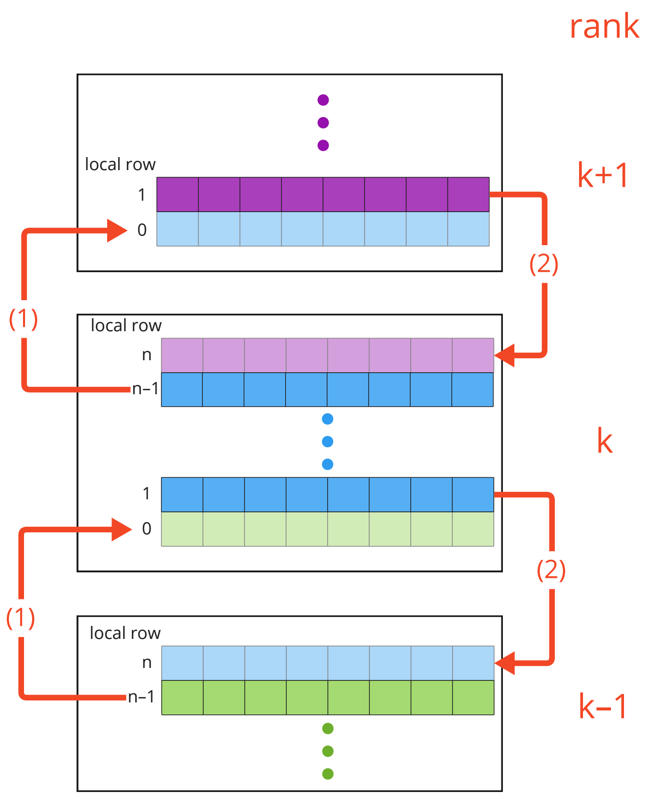

Fig. 2. Halo exchange for the kth rank.

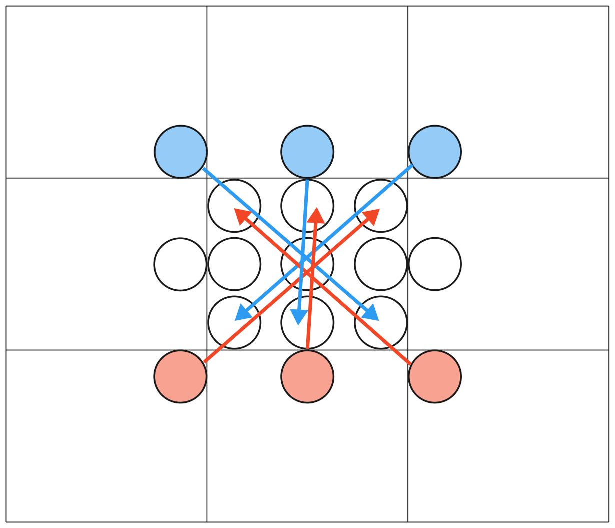

Fig. 3. The propagate step for a single cell in the grid. Squares represent cells and circles represent the speed values within a cell. Blue lines indicate data transfer from the row above, while red lines indicate data transfer from the row below.

iterate over different grid portions. As well as this, I also added a halo exchange after each completion of the accelerate flow step. This is best described with the aid of Fig. 2, which visualises the exchange for the kth rank. I used MPI's *MPI_Sendrecv()* function for message passing between the ranks. The exchange involves two steps. In the first, the kth rank sends a copy of the final row under its jurisdiction (row $n-1$) to its successor, while receiving a row from its predecessor, which is stored in its bottom halo region (row 0). In the second step, it sends a copy of the first row in its jurisdiction (row 1) down to its predecessor, while receiving a row from its successor, which it stores in its top halo region (row n).

There are several optimisations that I made for this halo exchange. Firstly, I included the halo regions as rows within the local grids. This ensured that when an array was malloced, the halo rows were adjacent to the assigned rows in memory. This takes advantage of the spatial locality used in caching, since adjacent data are fetched together, reducing the number of expensive memory accesses required. Since the LBM code is memory bandwidth-bound, this helps to improve the efficiency of the code during the *timestep()* function.

The second optimisation is that rather than copying a halo region to a separate buffer in some intermediate step before transfer, I provided a reference to the halo region itself in the *MPI_Sendrecv()* call. This saved on both additional memory and time taken for unnecessary copying of data.

Finally, I noticed an opportunity for further optimisation due to the nature of the propagate step. This is the only step within the simulation wherein cells transfer data to their neighbouring cells. As demonstrated by Fig. 3, each cell only receives the bottom speeds from the row above (blue) and the top speeds from the row below (red). Therefore, the only data that actually needs to be sent in the halo exchange are the top speeds when sending to the rank above and the bottom speeds when sending to the rank below. This greatly reduces data transfer, since we now only send $\frac{1}{3}rd$ of the speed data.

C. Reductions

Between each call of the *timestep()* function, I used *MPI_Reduce()* to perform a reduction to sum the total velocities within each grid. The sum is accumulated in the master (rank 0) process, which then solely calculates the average velocity for that timestep by dividing the sum by the total number of non-obstacle cells. This number is itself pre-calculated via a similar reduction during initialisation.

D. Gathering

In the collate portion that follows the main simulation, I use the *MPI_Gatherv* function to collect the data from each process into the master (rank 0) process. Using a similar optimisation to that of the halo exchange, the gathered data is stored directly into the process's own *all_cells* grid, which contains the entire final state of the grid in one process. The master process ultimately calculates the Reynold's number from this grid and uses it to write the results to the output file.

III. Results and Analysis

I ran the fully optimised parallel MPI code on all 28 cores of four nodes on BC4 (i.e., 112 total processes). A comparison of the total runtimes (including initialisation and collation) with those of the original optimised serial code is shown in table I. The result was a massive increase in performance, demonstrating the effectiveness of parallelising the code. One interesting observation is that the speedup increases with grid size. This is likely due to the relative utilisation of caching.

When running the optimised serial code, the program runs on one core of a single CPU. This only allows it to make use of the L1 and L2 cache associated with that core, as well as the L3 cache and DRAM for its CPU. However, the parallelised MPI code uses all 112 cores of four nodes. This makes full use of 112 lots of L1 and L2 cache (one set for each core) together with 8 lots of L3 cache and DRAM (one set for each CPU).

The data quoted in the subsequent paragraphs are quoted from [1] or calculated based on the malloced allocation used in the code (but are approximate since actual malloc allocations vary between ranks with different allocated rows).

First consider the 128x128 grid. The total malloced space is approximately 800 KiB for the optimised serial version, of which 32% will fit in L2 cache and the rest in L3. The malloced space in the MPI version is approximately between 29 and 38 KiB per process, which mostly fits in each one's L1 cache with some additional usage of L2. By comparison, with the 1024x1024 grid, approximately 40 MiB of space is allocated in the serial version. This is too large to fit in L3 cache (which has a capacity of 35 MiB on BC4's Broadwell architecture), so will have to make use of DRAM. Only 0.006% of this will fit in L2 cache. However, the MPI version makes full use of all available cache, such that around 30% of its total malloced space will fit in L2 cache, and using L3 cache it does not require DRAM at all.

Latency increases drastically with each level of cache (e.g. 3.4 ns in L2 compared with 16.3 ns in L3 or 55.9 ns in DRAM). This means that when the 1024x1024 grid is run using MPI code instead of in serial (shifting from DRAM to a significant usage of L2), the cache utilisation causes a proportionally higher speedup than when the 128x128 grid is run with MPI code instead of in serial (shifting from L3 to L1/L2).

Another possible explanation for the proportionally higher speedup on larger grid sizes is to do with communication overhead. On the 128x128 grid, every process is allocated 1-2 rows each (excluding the 2 halo rows). On the 1024x1024 grid, every process is allocated 9-10 rows each (excluding the 2 halo rows). This means that for the 128x128 grid, there is a much higher proportion of message passing per data processing (a 1:1 ratio at best of processed rows to transferred rows) compared with the 1024x1024 grid (a 1:5 ratio at best).

Table II shows a comparison between the runtimes when using 28 cores with the OpenMP shared memory model and 28 cores with the MPI message passing model. It is clear that the MPI version outperforms the OpenMP implementation on every grid size. Another notable feature of the results is that the MPI implementation shows greater speedup for the 128x256 and 256x256 grids than the others, with smallest speedup on

TABLE I
OPTIMISED SERIAL (OS) VS. MPI PARALLEL ON 112 CORES (MPI) PERFORMANCE.

Grid	OS Runtime (s)	MPI Runtime (s)	Speedup
128x128	22.26	0.47	47.36X
128x256	44.21	0.56	78.95X
256x256	176.41	1.42	124.23X
1024x1024	730.48	1.65	442.72X

TABLE II
OPENMP PARALLEL (OMP) VS. MPI PARALLEL (MPI) PERFORMANCE, BOTH USING 28 CORES.

Grid	OMP Runtime (s)	MPI Runtime (s)	Speedup
128x128	0.69	0.49	1.41X
128x256	1.18	0.69	1.71X
256x256	2.84	1.42	1.88X
1024x1024	14.51	11.18	1.30X

the 1024x1024 grid. There is therefore a correlation between speedup and simulation iterations, since the 128x256 and 256x256 inputs are set to have 80,000 iterations, while the 128x128 input has 40,000 and the 1024x1024 has the least number of iterations at 20,000.

This certainly appears to be counter-intuitive. Both implementations were run using the same resource allocation: 28 cores on one node of BC4. The only difference is in the approach to parallelism. The OpenMP version uses a memory-sharing model where each worker processes its set of rows independently every iteration. All the rows of the grid are stored sequentially in shared memory. Each worker also needs to read from some of the rows that are being worked on by adjacent workers.

By comparison, each process of the MPI version has its own individual allocation of rows that they store themselves. Rather than reading directly from the worker's neighbours' rows, it uses a halo-exchange message passing model to transfer this data.

It is possible that the message passing in the MPI implementation is more efficient that the OpenMP's memory sharing due to issues with cache coherence in the OpenMP version. In other words, since adjacent processes need to share access to the same memory, their L1 and L2 caches need to synchronise often, which could be causing additional overhead. Since the number of data transfers (whether that be halo-exchange or pointer swap) is proportional to the number of simulation iterations, this would explain why we see that correlation in table II.

Table III displays a comparison of the MPI runtimes on all 112 cores against the ballpark times. Again, these are notably faster and as with table I, the speedup increases with grid size.

Fig. 4 shows how the MPI implementation scales with respect to the number of processes (i.e. cores) that it utilises. The speedup (blue) increases rapidly until all 28 cores of one processor are utilised, at which point it begins to plateau. In a related fashion, the parallel efficiency decreases exponentially, ultimately dropping below 10%. It is also worth noting that there are negative spikes in both lines at 28, 56 and 84

TABLE III
BALLPARK (BP) VS. MPI PARALLEL (MPI) PERFORMANCE

Grid	BP Runtime (s)	MPI Runtime (s)	Speedup
128x128	1.7	0.47	3.62X
256x256	6.0	1.42	4.23X
1024x1024	14.0	1.65	8.48X

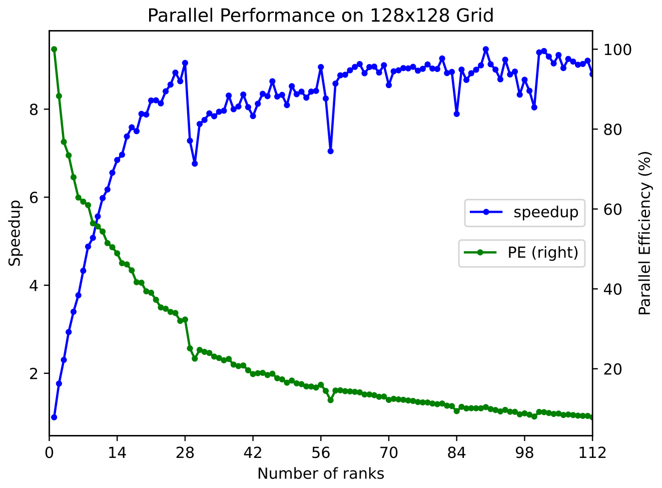

Fig. 4. Parallel scaling for the 128x128 grid.

cores, which corresponds to each node reaching its maximum capacity.

These drops in speedup are easily explained as being caused by communication latency in the halo exchange. For example, when the number of ranks is 28, all cores in the first node are being used for the simulation. Once 29 ranks are used, an extra core is required from a second node. The physical distance between this core and the two that it must communicate with in the halo exchange is much greater than the distance between any two cores within the same node, hence the higher latency and consequent drop in speedup. For each of these drops, there are at most two points on the graph for which speedup decrease before it starts to improve again. This makes sense; only two cores within the newly utilised node need to communicate with the previous node. Any further cores will just communicate with each other since the halo exchange takes place between adjacent processes.

The overall trend is sublinear scaling, which may be explained using similar reasoning to those based on the observations made with table I. As the number of ranks increases, the number of rows allocated per rank drops but the amount of message passing remains constant. Specifically, at 32 ranks, each process is allocated 4 rows and performs 2 row exchanges (to the rank above and below). At 64 ranks, every process is allocated 2 rows each and still performs the same 2 row exchanges. This ratio of processing to communication worsens with more ranks, leading to lower parallel efficiency, since the program is memory bandwidth-bound.

IV. CONCLUSION

The MPI implementation outperforms the optimised serial code by a strong margin and was also shown to have higher speedup than both the OpenMP version and ballpark times. Potential avenues for further optimisation include investigating a comparison between alternative MPI functions for message passing and data aggregation. One major untapped area is the final data collation, which could also be parallelised so that a reduction is used to calculate the Reynolds number and each process writes the final output individually, allowing for the collation step to happen in parallel.

REFERENCES

[1] "Xeon E5-2680 v4 - Intel - WikiChip," en.wikichip.org. https://en.wikichip.org/wiki/intel/xeon_e5/e5-2680_v4