Distributed memory parallelism of 2D lattice Boltzmann using MPI

Jarrod Doyle
ff18713@bris.ac.uk
1848668

I. Introduction

This report describes the steps taken to optimise runtime of the 2D lattice Boltzmann method (LBM) skeleton code provided by the unit. The target system of these optimisations is the University of Bristol's BlueCrystal4 (BC4) high performance computing system where the code is run in parallel on 112 cores distributed equally across 4 system nodes (28 cores per node).

All runtimes presented in this report are averaged across 5 runs. Table rows (excluding headers) can be assumed to relate to runtime on 128x128, 256x256, and 1024x1024 inputs respectively. An attempt was made to select the highest performing BC4 nodes to obtain the given runtimes (See section IV).

LBMs are a modern technique for simulating complex fluid systems. They are straightforward to parallelise and run efficiently on massively parallel architectures.

II. Previous Work

Earlier work produced for this unit optimised for serial performance as well as parallel performance using OpenMP. The code on which this report is based builds upon this previous work, making use of all its serial optimisations. This report therefore assumes general knowledge from the previous report, however a brief overview of the optimisations used and the performance achieved with them is also provided. Previous OpenMP code was not used.

Table II.1: Runtimes achieved at each optimisation stage in the previous report

Skeleton	Serial	Serial + Vectorised	OpenMP Parallel	Total Speedup
29.2s	16.6s	4.4s	0.9s	32.44x
233s	132s	42s	3.3s	70.61x
980s	569s	236s	14s	70x

A. Serial optimisation

The default C compiler on BC4 is GCC 4.8.5 and the skeleton code used the `-O3` optimisation flag. This was changed to ICC 19.1.3.304 with GCC 9.3.0 as the backend with `-Ofast` optimisation. After that the number of loops over the lattice per iteration was reduced from 4 to 2. Further loop fusion showed decreased performance due to inefficient memory access patterns decreasing preventing efficient vectorisation. Copying of data between `tmp_cells[]` and `cells[]` was removed by simply swapping pointers at the end of each iteration.

B. Vectorisation

The provided skeleton code was in an Array-of-Structures (AoS) form. This was converted to a Structure-of-Arrays (SoA) form. The `-xAVX2` flag was also used to produce an executable that makes full use of the AVX2 vectorisation processor instructions provided on the BC4 architecture. The c99 `restrict` keyword was used to explicitly inform the compiler that certain pointers could not alias. Finally, `malloc()` and `free()` calls were replaced with ICC boundary aligned equivalents and alignment compiler hints were used where necessary.

III. MPI Implementation

In the previous report a Single Instruction, Multiple Data (SIMD) architecture was used. In this report MPI is used, which is a Single Program, Multiple Data (SPMD) architecture. This means with MPI we have a single program which is run multiple times simultaneously, each of which is running on a different core and operating on different parts of the dataset.

A. Data Initialisation

An initial naive MPI implementation may initialise the entire lattice on each MPI rank and simple operate on a subsection of the lattice. However with large lattice sizes this is extremely memory intensive and slow. A more sophisticated approach is to allocate memory and initialise the lattice on each rank for only the subsection of the lattice that will be operated on. There are various options for the structure of the per-rank lattice subsections such as decomposing into tiles, by rows, and by columns.

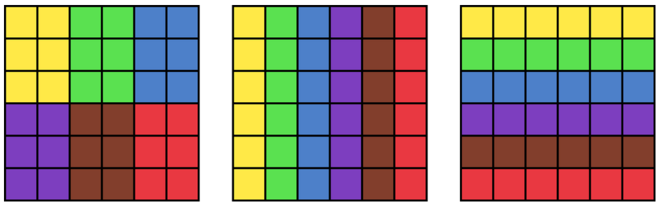
Illustration III.1: Example domain decomposition schemes. Left: Tiles, Center: Columns, Right: Rows.

The LBM code we are working with uses a 9-point stencil, processing each cell of the lattice requires information from the cell itself as well as the cells 1 step in each of the cardinal and intercardinal directions. This means that when working with a subsection, cells along the edges of the subsection will require access to cell data on another rank. Using an MPI message everytime a cell from another rank is referenced means each edge cell will need at least 3 messages (potentially more when using tile based decomposition). To avoid this, as well as storing the lattice subsection each rank stores a halo region which are extra cells around the edge of our section which are populated with data from neighbouring MPI ranks once each iteration. Due to the requirement of regular halo transfers the choice was therefore made to use row decomposition, as it allows for easier and more efficient packing and unpacking of halo messages. The halo exchange scheme is explained in more detail in section II.B.

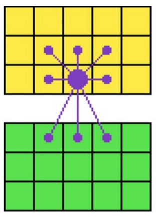
Illustration III.2: Cell stencil accessing cells from another MPI process

In order to balance the load given to each MPI process we attempt to give as close to equal amount of cells to each rank. When the height of the input lattice is cleanly divisible by the number of MPI processes this is simple, however in cases where there is remainder left over we must decide how to distribute what is left. Here it was chosen to distribute the remainder rows 1 per rank starting from rank 0. The same distribution strategy is used when loading the obstacle map. An exception here is that on MPI rank 0 an additional obstacle array is used that contains the entire obstacle map to avoid having to collate a final obstacle map later.

The changes to the initialisation stage lead to some additional information being stored in the `t_param` struct. These pieces of additional information are the number of MPI processes being used, the rank of the current MPI process, the ranks of the top and bottom neighbours, the start and end y values of the lattice subsection for the rank, and the number of rows in the rank.

B. *Computation*

Due to each process now operating on a limited subset of the original input lattice some changes have to be made to the computation phase. The first of which is that flow acceleration only occurs in a single rank. The skeleton code performs flow acceleration along the second row of the lattice, our use of row based lattice decomposition means that the accelerated row can only be in the local cells of one MPI process meaning we can skip this step for all but one rank. In some cases the accelerated row lies in the halo region of another MPI process. One option would be to also accelerate the row here, however our next choice means this isn't necessary.

After accelerating flow we then perform a halo transfer. There must always be a halo transfer on each iteration and by performing it after flow acceleration any ranks that contain the accelerated row in their halo region will receive the accelerated values.

One of the most important parts of implementing MPI is ensuring a good communication pattern between ranks and avoiding deadlock where every rank is blocked waiting for a message to be sent/received. The method used here makes use of the `MPI_Sendrecv` API call which safely performs a send and receive at the same time without having to worry about order. We do however still have to ensure that there is both a send and a receive to be processed. To do this two `MPI_Sendrecv` calls are used. The first receives an updated halo region from the rank below and sends an updated halo region to the rank above and the second does the opposite, receives from the top and sends to the bottom. This avoids deadlock by ensuring there is always a matching `send/recv` pair.

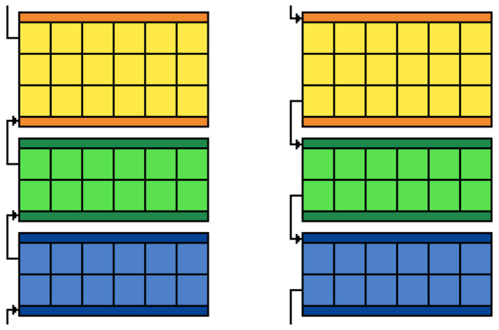
Illustration III.3: The two stages of the halo transfer scheme. Left: Receive bottom send up, Right: Receive top send bottom.

After both directions of halo transfer is called we propagate our speeds. This is essentially unchanged from the pre-MPI code, however we ignore the first and last row as these contain our halo regions and don't need to be processed. Our `tmp_cells[]` therefore doesn't contain accurate halo region values, but rather the halo region values from the previous iteration. This isn't a problem as we don't need to read from the halo region again until this same time next iteration, at which point the halo region in `cells[]` will have been updated with values from neighbouring ranks.

The final changes are to the collision function. Here we again ignore the first and last row, this time in `tmp_cells[]` rather than `cells[]`. The other change made to collision is that it no longer returns the average velocity, rather it returns the total velocity of the local cells of the rank. This change is made because working local averages don't translate well to a global average velocity. Global average velocity is now calculated by a single rank during the collation stage.

C. *Result Collation*

At this stage cell and velocity data is still spread across the MPI processes. Before outputting we must first collect that data. To do this one rank is chosen to be the collection point, rank 0 is a standard choice and the one used here.

MPI provides numerous methods for broadcasting and collecting data to/from all ranks, these are known as collectives. Here the goal is to collect information from all ranks to one rank, so we want to use a gather. However as discussed earlier it's possible that ranks have uneven amounts of cells and therefore a variable gather call `MPI_Gatherv` must be used. It isn't required to do an additional gather to work out the size of each rank, they all have one of two sizes. If the rank index is less than the remainder of input height modulo the number of processes, the size is number of rows in the rank plus 1 multiplied by the width of the input. Otherwise, it is just the number of rows in the rank multiplied by the width of the input. In practice however I saw no difference in runtimes between calculating the sizes manually on rank 0 and doing a gather and having each rank send it's cell count. Using the build variable gather size list we can then collate the final cell speeds on rank 0.

The last thing to collate is the average velocities at each timestep. Remember velocities on each rank are now stored as a total of the velocities on the ranks local cells, therefore we first collect velocities on rank 0 using an `MPI_Reduce` call with a sum operation giving a global lattice velocity at each timestep. A total cell count is then calculated by which the velocity at each timestep is divided to give the final average velocity for each timestep.

IV. Performance Analysis

Table IV.1: Runtime comparison of serial and MPI parallelised code

Serial + Vectorised	**MPI Parallel**	**Speedup**
4.4s	1.2s	3.7x
42s	2.7s	15.6x
236s	3.0s	78.7x

One important thing to note here is that drastically different runtimes were observed depending on which BC4 nodes were being used. On the 128x128 input the lowest runtime observed was 0.6s, and the highest was 3.1s. On the 1024x1024 input the lowest runtime observed was 2.65s and the highest was 7.8s. Experimenting with the available BC4 nodes showed that a more consistently fast performance could be achieved by using a combination of compute nodes 100-102 and 105. This is therefore the selection of nodes used for comparisons.

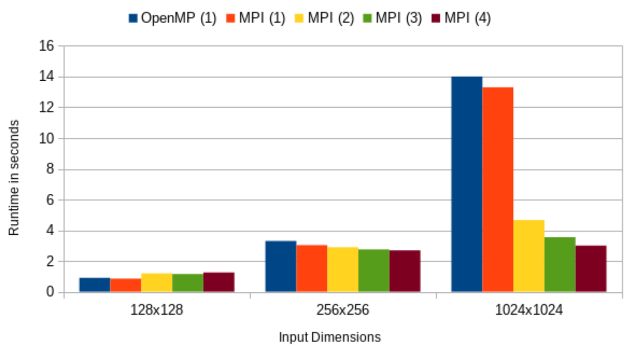

Illustration IV.1: OpenMP and MPI runtimes. Numbers in parentheses show the number of full BC4 nodes the code was run across

Initial comparisons to vectorised serial performance show improved runtimes, however in comparisons to the previous OpenMP implementation the MPI version had decreased performance on 128x128 inputs, slightly improved performance on 256x256 inputs, and much better performance on the larger 1024x1024 input. The lack of improved performance on the smaller input sizes is mainly due to the ratio of halo region cells to local region cells being very high. For example, with 4 nodes being used there are 112 processes which means each row is in at least one halo region and a majority are in two halo regions. This means in each iteration more cells are being transferred during halo transfer than are being processed. This also provides some explanation the relatively small 3.7x speedup gained over the serial code on the 128x128 input. These smaller inputs are heavily bound by memory bandwith rather than compute ability.

V. Conclusion

Table V.1: Achieved MPI runtimes and the course provided target times

MPI Parallel	**Target Ballpark Time**
1.2s	1.7s
2.7s	6.0s
3.0s	14.0s

Parallel runtimes using MPI improved vastly compared to serial + vectorised and beat the target ballpark times provided for the unit. Careful planning of the MPI architecture before implementation allowed for a relatively painless implementation process with good results. Decreased performance on smaller input resolutions relative to the previous OpenMP implementation was an initial surprise but made sense once the ratio of halo region to local region was considered. Further improvements may have been possible by experimenting with a combination of MPI and other parallelisation techniques such as OpenMP or OpenCL. This is something I would have liked to experiment with however BC4 was experiencing issues during the time I'd scheduled to try this.