High Performance Computing MPI

Sunny Miglani
sm15504

May 13, 2018

1 Introduction

The aim of this report is to document optimisations and improvements on an implementation of the Lattice Boltzman method. This report discusses the optimization done with MPI to distribute the computation over multiple cores on the BlueCrystal Phase 4. The optimisations are focused on a 1024x1024 grid, but the report shall include sizes 128, 256 for contrast as well. This implementation uses a structure of cells to represent the velocities of the points in the grid with 8 different directions. A requirement for each step of the program is that each cell knows the surrounding eight cell's values to be able to update it's own.

2 Initial Timings and Simple Optimisations

The given implementation of the Lattice Boltzmann code written in C runs at the times shown in Table 1, the table shows the vanilla timings with no optimizations, and the timings after using a more focused *icc* compiler with $O3$ and $fast$ flags.

As the focus of this report is on the MPI optimisations and the MPI structure, the serial optimisations have been omitted from the report.

Sizes	Runtime Vanilla(s)	Runtime ICC(s)
128x128	31.90	28.07
128x256	63.07	55.80
256x256	257.89	216.73
1024x1024	1101.11	1020.94

Table 1: Vanilla Timings and icc timings

3 MPI Structure

MPI allows the program to be split amongst n workers over the network of cores. This means that the method of splitting this problem is quite important to the efficiency of the program.

This efficiency can be measured by the communication overhead in the program. The lower the communication overhead, the more efficient the system can be thought of as the slowest part of any concurrent program is the communication between workers.

The selected implementation for this report is the method of horizontally splitting the grid of cells. This system allows each worker to work independently of each other once an extra row of cells, called the "halo", has been added to the top and bottom edges of the worker's sections. The halos are needed as the propagate function requires that each cell know it's surrounding eight cells for some crucial calculations. These halos are updated by their neighbouring cells to make sure each worker has the most up to date version of their neighbours, and sending only the halos keeps the amount of data communicated at a minimum. Halos also prevent the need for the program to access some form of shared memory between its workers.

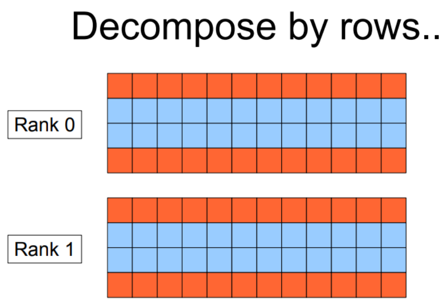

Figure 1: Visualisation of Splitting with Halos

In Figure 1, we can see a trivial example of the structure, where the red cells indicate the halos for the workers. The concept of a halo exchange is quite common in these grid based concurrent implementations.

3.1 Breaking down the grid

- Each worker initializes the data themselves, allowing for quick initialization and reduces the need for the initial communication with the master thread.
- Each worker initializes the entire grid space as opposed to an offset just for themselves. The program then finds the limits of a "working zone" and their halo rows by measuring the indices they are meant to work over using their *rank*. While this is not the most efficient implementation in terms of memory, it means that the rest of the program's functions do not require much change therefore keeping the correctness of the program constant. This system also retains the flow of the program making it easier to debug and keep under control.
- Since the workers know their indices for the working zone and the halo exchanges, they can proceed to work on the iterations without the master's input. The only communication with the master therefore is for sending the average velocity values and after the iterations have been completed over the grid for the final result.
- This system allows the communication to be focused on the workers, which use $MPI_Sendrecv()$ function. $MPI_Sendrecv()$ can be run concurrently amongst the programs, therefore keeping the time held on communication to a minimum.
- It is also good to note that the master is involved in this process of calculation, making sure that there's no thread that's idle during the computational steps.

3.2 Dependency on Workload Distribution

The way the current workload of the grid is distributed amongst the workers is using the system described in the Section 3.1. For a non divisible split in rows ($row\%worker \neq 0$), the last worker gets any extra rows that are left over. This means that the last worker is forced to do more computation than the rest of it's cohort. This causes a slowdown over the entire cohort in the halo exchange step, especially on the first and second last worker, as they heavily depend on the last worker.

An optimisation on this system is discussed in Section 6.1

3.3 Working with Average Velocity

The average velocity calculated in each iteration is used for error checking the implementation to make sure that scaling it does not cause any problems with the accuracy of the simulation. Due to the method in which the average velocity is calculated, the broken down structures and workers simply calculating their velocities would result in an incorrect value, creating problems in the error checks. To fix this implementation, the system uses the MPI's function of *Reduce*() that can apply an operation over the whole cohort of workers for a particular variable.

In the case of average velocity, this *reduction* is applied on the summed values of the velocities of the non obstacle cells. Using the reduction with a sum operation, we give the master the total velocity of each worker, and the master thread can then calculate the overall average velocity for the cells and save them for later error checks.

Using the *Reduce*() function allows for a slightly faster communication as it makes each program communicate with the other concurrently rather than forcing the programs to communicate sequentially with the master.

3.4 Gathering Data for Reynolds' Number

The final step in the MPI implementation is to gather the data from the workers after running all the iterations so that the master thread can calculate the Reynolds number needed for the simulation. To gather this data, simple $MPI_Ssend()$ and $MPI_Receive()$ calls which are part of MPI are used.

4 Timings of the Implementation

A good MPI application or implementation would allow scalability (upto a point) with an increasing number of nodes. This application has multiple problem sizes that can be tested for scalability and is therefore a good test for the implementation done in the program.

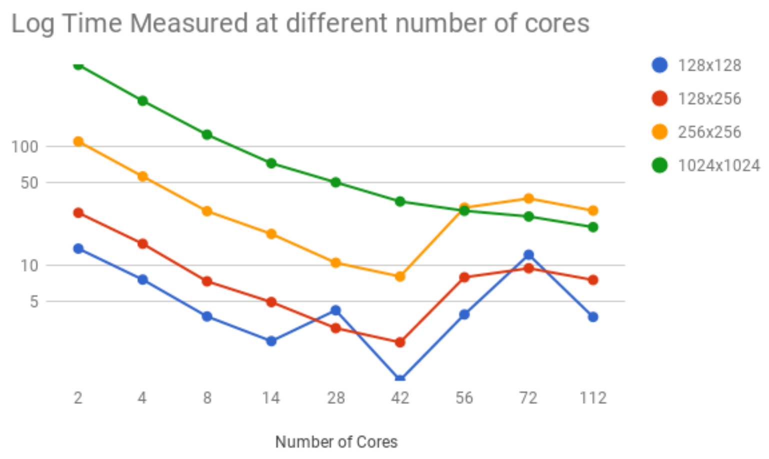

Figure 2: Visualisation of Splitting with Halos

Figure 2 shows the timings of the MPI implementation over various number of nodes. The Y Axis is the time in seconds in a log scale making the scaling of the values comparable through the different sizes.

It is visible that there is linear scaling with the increase in number of cores. This trend is seen until different points for different sizes of the grid.

For smaller sizes we see that beyond the 42 core mark, the time taken actually increases, which can be attributed to the fact that beyond 42 cores, there are only 2-3 rows to be worked on per core which makes the program more communication dependent rather than computation dependent. This means that the time taken measured is not dependable on multiple iterations as it would depend on the distance needed to travel for each worker (how the cores are distributed amongst nodes).

For the larger size of 1024x1024, we see a linear scale that goes on and the time taken decreases as the number of cores increase, this shows that the program in these larger cases is more computationally bound, which means that the time is dependent more on time taken for computation, and less on communication.

It is possible to calculate the threshold after which the program is no longer computation bound but rather time bound. The threshold would depend on the number of rows given to each worker while the number of columns is left constant. For example, for the threshold for the 128x128 sized problem is 3 rows with 128 columns each. Beyond this we see a spike in time taken as the program switches from being computationally bound to communication bound.

5 MPI Final Timings

With the current optimisations and implementation method, these are the final timings obtained at one node (28 cores) in Table 2.

Sizes	Runtime(s)
128x128	4.16
128x256	2.94
256x256	10.51
1024x1024	50.26

Table 2: Final Timing on the MPI implementation

6 Future Optimisations

There are many different implementation optimisations that can be made in the program, some of these are listed below:

6.1 Balancing Workload

With the current workload split across the threads in a set manner and giving any extra rows to the last worker, we get an unfair split amongst the cohort as mentioned in Section 3.2, to fix this a simpler system would be taking any extra number of rows and spreading that in a round robin fashion amongst the workers. The idea being that this would reduce the workload on the final worker, making the process of halo exchanges more balanced.

6.2 Merging 'for' loops

A big bonus to any grid based program is the ability to vectorize over the loops in the program. In the current implementation of MPI, these for loops contain *if* statements / conditionals that would prevent automated vectorization through the compiler. To give the compiler the space to vectorize the for loops, the following changes would have to be made:

- Move everything into one large for loop over iterations.
- Remove conditionals, and if needed, use ternary operators "? :".
- Remove the constant accesses to shared structures, and replace it with local variables in the loops.

These are the first steps to try and vectorize the program enough that it could run to a much higher speed.

6.3 Asynchronous Halo Exchange

The current halo exchange requires all the workers to sync up at the same point, and concurrently transfer data between them. Alternatively, due to the independence of the inner cells (cells not next to halos) in relation to the halo, the program should be able to proceed for one iteration through in the inner cells of the grid while the program waits for a halo transfer. However, this would only give any worker the advantage of one iteration after which it would be waiting for the halos to be updated.

7 OpenMP implementation for GPU

An alternative extension is making the code run on a GPU instead of a CPU as GPUs are often much better at handling structured data such as an array. The GPU allows a more efficient version of Single Instruction Multiple Data which should increase the performance of the program by a large amount.

The GPU implementation referred to in the following section uses an implementation with simple *omp pragma* commands that treat the GPU as the *target*. This GPU implementation does not include MPI and is therefore running on just one graphics card.

Some vanilla timings with the GPU are shown in Table 3.

Sizes	Runtime Vanilla(s)
128x128	31.07
128x256	61.01
256x256	235.45
1024x1024	931.25

Table 3: Vanilla Timings on the GPU

To increase the speed of the GPU implementation, having a system of merged for loops mentioned in 6.2 would allow for vectorization on the GPU as well. Since the GPU heavily depends on SIMD, a vectorized loop would create a quite drastic speedup in the program allowing for a future integration with MPI to allow multiple GPUs to run the code at the same time.

On comparison to the vanilla times shown in Table 1, we see that the GPU already has an advantage in terms of computation. In contrast however, any data transfer of data to a GPU takes a lot more time. This means that deciding whether a program should run on a GPU or a CPU depends not only on the type of the problem, but it's reliance on communication between the parallel workers.