Optimisation of d2q9-bgk Lattice Boltzmann Scheme with MPI

James Elgar, za18968

May 8, 2021

1 Introduction

This report will explore the use of distributed memory parallelism to optimize a given algorithm. The algorithm solves a d2q9-bgk Lattice Boltzmann scheme over a grid of cells. The report will evaluate different approaches to distributed memory parallelism, mostly using the Message Parsing Interface, MPI, standard. When optimizing a distributed memory problem there are various factor which can effect the performance of the algorithm. This report will focus of the optimization of individual nodes, the data layout of the cells array, the use of blocking vs non-blocking sends and final compare the use of MPI for running with multiple processor with OpenCL for running with a graphics card.

2 MPI

The provided algorithm was optimised using shared memory parallelism with OpenMP. This means every thread or core has access to the same region of memory which is manipulated during the calculations. This approach avoids having to share sections of memory between different processors and can keep the implementation simple. However this approach does have limitations. When scaling up to using multiple CPUs, all CPU would be using a shared bus for memory access. This can dramatically effect performance of memory access especially when the number of CPUs is high. To prevent this distributed memory parallelism can be used. This is where mutliple regions of memory are used and the required synonisations of the data are made. This allows each CPU to use it own cache and thus can improve the performance of memory access.

In this MPI implementation 4 nodes were used. In order to distribute the work across these 4 different nodes, the grid was divided up into 4 sections. When dividing a grid into different sections there are generally 2 approaches. One approach is square sub grids as shown in Figure 1 and the other dividing into groups of rows, as shown in Figure 2. In both approaches halo regions are shared between the two nodes to synonise the required sections of memory. For this problem each cell requires all the cells around it and therefore any edges require a halo region. The squares approach minimises the amount of data which has to be shared, as a square has a small surface area relative to its permimeter. MPI can send sections of contigious data between ranks, however if the data is not contigious then either multiple sends are required or the data must be made contigious first. This means the smaller surface area is counter acted by having to send more regions of memory seperately as each of the 4 sides of the square are not contigious. For this reason the rows approach was chosen for this implementation.

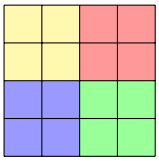

Figure 1: Tiled distribution

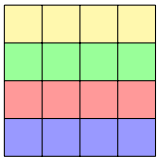

Figure 2: Rows distribution

A closely related consideration to the shape of the rank regions is what data is shared between the ranks and when the data is shared. Figure 3 shows the halo regions which must be shared between each rank every timestep. In the inital implementation `MPI_Sendrecv` was used to send these regions at the begining of each timestep. This is a blocking operation, meaing the execution is blocked until the send and recieve for that node have been completed.

The results for this computation on 4 nodes using mpi compared to a serial implementation are available in **??**. This shows the relatively minor improvement in performance even when the work is divided across the 4 nodes. This is as a result of the blocking operations. At the begining of each timestep each rank must wait for all the data to be send and for the ranks above and below to send the halo regions. This can dramatically increase the run times and result in idling time where the CPUs are just waiting for data to be send.

3 Non-Blocking Send

MPI offers methods to avoid this wasted CPU time by enabling non-blocking communication. This allows the CPU to asynchronously send and receive data to and from a rank in the background, allowing the CPU to continue with other computation in the meantime. The implementation of non-blocking sends in this case required 3 MPI directives.

- `MPI_Isend`
- `MPI_Irecv`
- `MPI_Wait`

`MPI_Isend` and `MPI_Irecv` allow MPI to start sending and receiving data respectively but does not wait for these operations to complete. An `MPI_Wait` is then required to ensure the data transfer has completed. algorithm 1 shows the implementation of each timestep. The key difference between this implementation and the blocking one is the communication. At the beginning of each timestep the send and receives are initialised. Next the calculations for the inner cells, those which only depend on cells stored in local memory, starts. This allows the cell transfer to take place in the background whilst this computation is happening. After the inner cells have been calculated an `MPI_Wait` is used to ensure the communication has finished. Finally the 2 outer rows are calculated using the cells received during the communication.

This approach allows the majority of the communication to happen whilst the others cells are being calculated. This

dramatically reduces the wasted CPU time, waiting for all the rank to be read and the communication to be completed.

Algorithm 1: Timestep implementation

```
1 Start receiving halo cells using MPI_Irecv
2 Start sending halo cells using MPI_Isend
3 for i ← 1 to ny − 2 1 do
4 |   Calculate next timestep cell values
5 end
6 Wait for alls sends and receives to be completed with
    MPI_Waitall
7 Calculate next timestep cell values for row 0 and ny -
    1
```

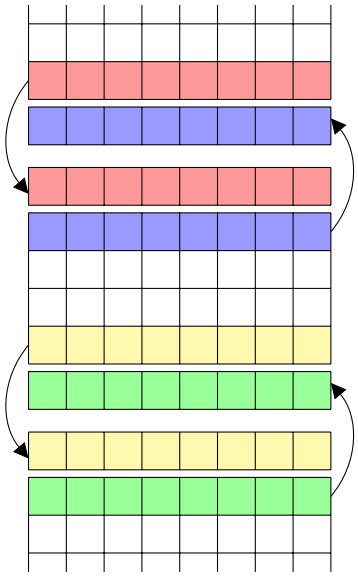

Figure 3: Cell regions and halos

4 SOA vs AOS

When sending data between ranks with MPI another important consideration is the local data layout. When using MPI sends the data must be contiguous. This means the cells being sent must either be stored in contiguous memory or be made contiguous before the sends. This means it is often faster to store the data in such a matter than it can be directly sent without any preprocessing. However another consideration of performance is the optimizations of individual nodes. The provious report discussed the importanance of memory layout for serial optimizations. The most significant case being vectorisation, where the CPU can use vecotr registers and instructions to dramatically reduce the number of instructions required.

In many cases therefore a tradeoff has to be made between optimizing the rank serial runtime and optimizing the time to send communications. In this case 2 approaches were used for storing the cells array.

- Sotring the cells as an array of strucutres, where each strucutre is 9 floats
- Storing the cells as a strucutre of arrays, where the strucutre stores an array for each speed

The first approach, array of strucutres, required only 2 send per timestep, the top halo row and the bottom halo row. These sends are acheived by using a custom `MPI_Datatype` and an offset. This means the send if very efficient as no additional dataprocessing is required before the send.

In contrast when using the second approach, array of structures, 6 sends are required (or the data must be arranged first). This is because each array is not contiguous in memory and therefore an offset cannot be used to send multiple speed arrays at once. When tyring to minimise the overhead of communication, the first option would therefore be the clear choice. However as shown in **??**, the runtime when using this approach is far slower. This is because each rank is no longer vectorised and thus each rank takes a longer time to complete each timestep.

This effect is also mitigated by using non-blocking sends. When communication is able to happen concurrently along side computation, the increase time of sending and recieving data can have minimal impact on overall run time.

5 OpenMP MPI

Although MPI uses distributed memory parallelism, which reduces run times by sharing the workload across multiple CPUs, it is also important to optimize the individual nodes. In this case it is possible to combine shared memory parallelism, on each rank, with distributed memory parallelism. Using OpenMP the loop for the inner cells in algorithm 1 can be parallelised. This makes use of all the threads in each node resulting in a dramatic speed up.

This topic was discussed in more depth in the first report.

Frontal face detection results		
Problem Size	TPR	F1-SCORE
128x128	601.7	1
128x256	601.7	1
256x256	1	0.88
1024x1024	1	0.667
dart14	1	0.5
dart15	1	0
Average	1	0.609

Table 1: Blocking vs Non-Blocking Sends

6 OpenCL