HPC MPI Coursework

Luca Gough - Student Id: 2108765

April 2024

1 Introduction

This project provides an analysis of the performance optimisations for the D29Q Lattice Boltzmann fluid simulation, a structured grid problem which performs some computation on each cell per turn. Each cell requires information from it's neighboring cells implying any parallel implementation will require some synchronisation between iteration steps. This project specifically analyses the distributed memory parallel approach which follows the single program multiple data paradigm (SPMD).

2 MPI

2.1 Theory

For this project we will analyse the program's performance scaling over 4 Blue Crystal Phase 4 nodes. Each node contains a dual-socket CPU, each socket contains 14 logical cores. This enables us to split to problem across $4 \times 2 \times 14 = 112$ cores and process in parallel. Amdahl's law defines a theory for computing a theoretical maximum speedup given a number of processes.

$$p = \text{Proportion of runtime that is parallelisable} \tag{1}$$

$$S = \text{Theoretical Maximum Speedup} \tag{2}$$

$$\lim_{x \to \infty} S = \frac{1}{1-p} \tag{3}$$

$$S_n = \frac{1}{(1-p) + \frac{p}{n}} \tag{4}$$

If we assume that parallel portion of the runtime reduces to zero as the number of processors tends to infinity then the serial sections of code become significant. Profiling the code enables us to compute the proportion of the runtime that is parallelisable and therefore we can compute a maximum theoretical speedup up of 101x across 112 cores.

2.2 Distributed Memory Model

The single program multiple data model describes a single program that is distributed across multiple compute nodes. Each 'rank' follows a distinct control flow which typically will initialise it's own data structures, share computation results with other 'ranks' after each iteration and finally gathering results on a single 'rank'. Each rank accesses it's own private memory space and must communicate asynchronously via some network interface. For this project the network interface is provided by the MPI standard which defines a set of implementations for point to point communication. Note each of the optimisations described in this project build on top of the optimised-vectorized code which utilises instruction level parallelism (SIMD). Additionally we enforce the Non-Uniform-Memory-Access awareness described in the single node parallelism version of this project. Each MPI rank is bound to a specific core using 'srun –cpu-bind=cores', ensuring that task's associated data is stored in the most local memory and therefore minimises higher latency memory access over the socket-socket interconnect.

After profiling the single node multi-core program using Intel VTune, we can determine that the largest performance bottlenecks occur at sections of synchronisation in the code. This is because some cores may be waiting for other cores to finish their current iteration step computations caused by a workload imbalance between threads. Another source of synchronisation delay which becomes apparent in a distributed system is the communication latency between two nodes. For these reasons we are unable to maximise processor utilisation and therefore it becomes imperative to minimise both workload imbalance and communication latency.

2.3 Rank Communications

2.3.1 Halo Exchange

In order to minimise communication latency we opt to utilise a halo exchange system, instead of distributing the entire grid to every rank at the end of each iteration. The halo exchange transfers the information at the boundary between each rank's subset of the grid. Additionally this information is only transmitted to the rank that requires it for the next iteration. This means that as the grid size $N \times N$ increases, the transmission delay increases on the order of $O(N)$ instead of $O(N^2)$. Additionally we must determine whether to the split the grid row-wise or column-wise between the ranks. Since MPI requires each message to be stored in a contiguous buffer, column-wise partitioning would require the program to pack and unpack the data into a 'send' buffer.

Grid size N	columns (s)	rows (s)	speedup
128	0.75	0.58	1.29x
256	4.62	3.41	1.35x
1024	20.46	17.75	1.15x

Table 1: **Column vs row communication:** This table demonstrates the impact of column-wise vs row-wise partitioning on runtime speedup.

Table 1 shows the effect on runtime for different problem sizes using the different grid partitioning schemes. Note that this experiment was performed on a single Blue Crystal Phase 4 node (28 MPI threads) due to availability issues, however the data packing routines remain the same. This delay is resultant from the overhead of the data packing functions.

2.3.2 Workload Imbalance

In order to address the workload balance issues we must distribute an equal amount of work to each rank. This corresponds to equal number of grid rows, however if the grid size is not divisible by the total number of MPI threads then we must define a routine for distributing the leftover work. The simplest approach would be too allocate the remaining rows to the final rank however this would generate a large performance bottleneck as the first $k-1$ ranks will finish their computation before the kth rank, introducing a section of serial execution as the threads wait to synchronise. In order to address this issue we can split the remaining rows across the ranks, allocating an extra row to each rank until each row is accounted for. This means each rank will wait for at most one row to process. Increasing the simultaneous utilisation of each core. The effect of this optimisation can be observed by profiling the program with intel Vtune which reports the total MPI spin wait time accumulated across 56 cores on two nodes. The 'MPI Imbalance' metric shows the CPU time spent by ranks spinning in waits on communication operations, normalized by the number of ranks, non optimal communication schema will result in higher MPI Imbalance values. Distributing the remaining work between all ranks halves the MPI imbalance.

2.3.3 Interleaved Communication + Processing

We can further reduce the time each rank spends waiting for the completion of communication operations. This is achieved by allowing the ranks to perform some work after transmitting their first and last rows, but before receiving the corresponding rows from it's neighboring ranks. This implies that instead of spin waiting we can spend that time doing useful work for the next iteration. We can leverage the fact that only the boundary rows require information from other ranks so we can compute the 'propagation' step for all the internal rows whilst waiting to receive the halo regions. Figure 1 shows the communication between 56 ranks, (red implies longer delay, green implies a shorter delay). We can observe that ranks with neighboring indices are transmitting their halo regions. The left diagram highlights this delay with a blocking communication schema, in contrast the right diagram represents the delay with interleaved communication and grid processing. This delay is reduced because the information will have already arrived at each rank before it calls the 'receive' function.

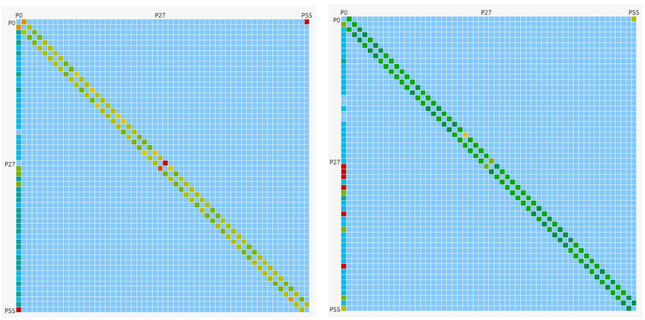

Figure 1: **Rank to Rank Communications:** The left diagram shows the communication delay between ranks without interleaving communication and processing. The right diagram corresponds to the communication of the program with the optimisation

On the 1024×1024 grid this optimisation increases the average CPU utilization by 15%, from 18/28 cores per node to 22/28 cores per node and provides a 1.45x speedup to the runtime. This optimisation is also fundamentally limited by the latency of the message passing. The maximum reduction in runtime is equal to the total amount of time spent in communication and therefore on larger grids this optimisation has a larger effect on the speedup, achieving 1.51x on the 2048 grid.

2.3.4 Result Gathering

Once each rank has finished processing the grid for N iterations we must gather the results on a single rank enabling it to write those results to a file. This will add a significant delay to the program as every rank will need to send a message to a single rank. This delay can be seen in figure 1 as an increased latency on the left column. Depending on the implementation 'MPI_Gather' should reduce this overhead by minimising the depth of communication operations performed on the root rank. Instead increasing the span by performing intermediate collections on other ranks. Finally we can utilise 'MPI_Gatherv' which enables us to send varying length buffers by specifying a displacement from the start of the global buffer.

3 Evaluation

3.1 Compute Scaling

Figures 2 and 3 demonstrates the program speedup scaled from 1 core to all 112 cores for the 2048 and 1024 grid sizes respectively. Experiments exhibit near linear scaling (indicated by the green line) for an initial number of cores before decaying into sub-linear scaling at the higher core counts. This occurs because as we increase the number of MPI threads the number of communication synchronisations increases and therefore the average simultaneous CPU utilisation decreases due to increased total wait time. Additionally this increases the result gathering overhead due to increased message passing. This decay is reduced by employing the interleaved processing and message passing because this will diminish the communication overhead.

Initially the 1024 experiment 3 demonstrates super-linear scaling, this likely occurs as each thread is assigned a diminishing proportion of the work it can fit the data in the smaller but faster-to-access cache levels. This assumption is reinforced by the fact that the 2048 2 grid doesn't exhibit this behaviour because the work groups can never get small enough to fit entirely in the lower cache levels. This effect enables the 1024 grid to remain in the linear scaling regime for much longer (60 cores instead of 25). The 2048 grid experiences much larger noise between runtime measurements, this is partially due to the fact that the larger grid requires more data to be transmitted on each iteration. This therefore adds more traffic to the network which may induce some latency to the data transmission depending on if the network bandwidth is saturated.

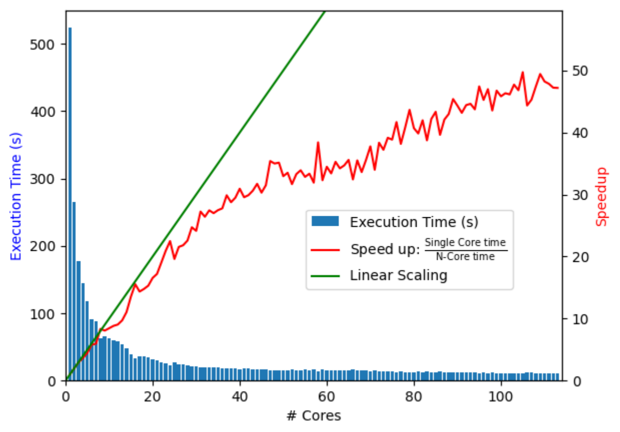

Figure 2: **Input 2048x2048 scaling:** Speedup scaling from 1 core to 112 cores across 4x28 core BCp4 nodes on the 2048x2048 input grid.

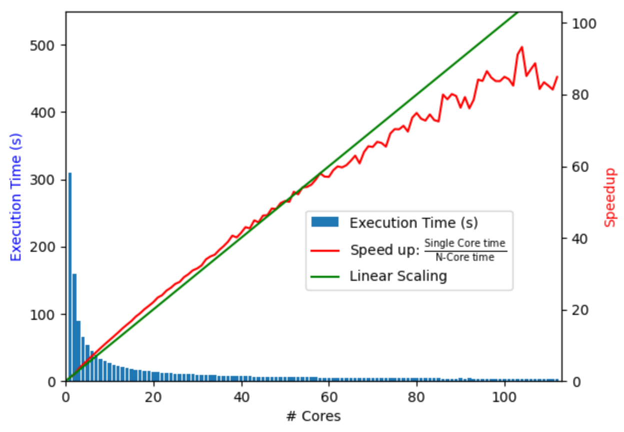

Figure 3: **Input 1024x1024 scaling:** Speedup scaling from 1 core to 112 cores across 4x28 core BCp4 nodes on the 1024x1024 input grid.

3.2 Problem Size Scaling

This section analyses the program's runtime with respect to the problem size. Specifically table 2 demonstrates the runtime per unit area of the input grid, this provides a metric for how well the program scales with increasingly larger grids. A larger value indicates a higher runtime per unit area. Therefore the 2048 side length grid results in the lowest runtime for a given grid size. This result occurs because all the grid sizes exhibit a similar collation time at the end of the execution. Additionally in all four experiments the same number of messages are sent (even if they differ in size) which incurs a similar communication overhead, this essentially provides a lower bound for runtime.

Grid Size: N	Runtime (s)	$\frac{\text{Runtime}}{N^2}$
128	0.89	5.43×10^{-5}
256	1.80	2.75×10^{-5}
1024	3.15	0.30×10^{-5}
2048	10.87	0.26×10^{-5}

Table 2: **Problem size scaling:** This table highlights the runtime per unit area of the input grid on 112 cores (4x BCp4 nodes).

3.3 Roofline analysis

A roofline analysis provides an evaluation of a program's performance compared to it's operational intensity, this data is presented in the context of the hardware limitations, specifically maximum floating point performance and memory bandwidths. Figure 4 shows a roofline diagram for the Lattice Boltzmann code (due to availability issues we run these experiments on 2 nodes totalling 56 cores), since this code is memory bandwidth bound we can observe the program's immediate rooflines are the L3 and DRAM bandwidths. This indicates that the program is able to take advantage of the higher bandwidth cache levels in order to fetch data at a higher bandwidth than the maximum achievable by DRAM. Additionally we can observe that the program is able to better leverage the faster caches when operating on the 1024x1024 grid. This occurs because the CPU is able to fit more of the data in these smaller caches and therefore reducing the number of high latency cache misses. The fact that all experiments exhibit a higher GFLOP/s than the peak for single precision scalar adds implies that they all make good use of vectorization. The 2048 grid running on 56 cores (2x BCp4 nodes) runs at a much lower operational performance. This reinforces our result from figures 3 and 2 where the 2048 grid transitions into sub-linear scaling at a much lower core count. This is resultant from the increased communication bandwidth requirements since we don't see this result when running on a single core. Additionally the 2048 grid requires twice the amount of memory loads and since a memory load typically takes more cycles than most floating point operations we can expect to see this reduction in FLOP/s.

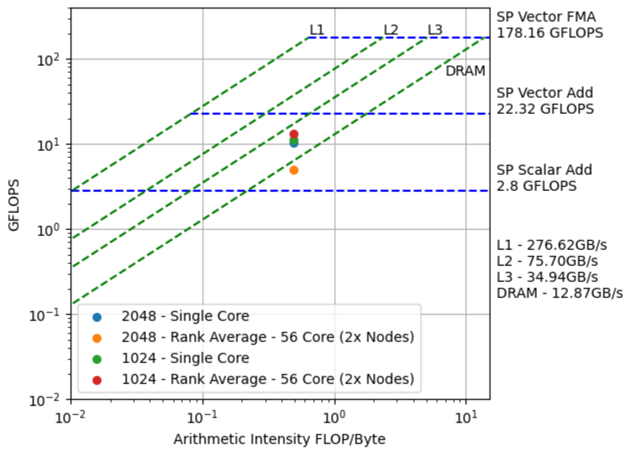

Figure 4: **Roofline Graph:** This graph shows the roofline analysis, comparing the performance of the single core program with average across all ranks of the 56 core program on the 2048x2048 grid.

4 Conclusion

According to Amdahl's law we should achieve a maximum theoretical speedup of 101x on 112 cores. The maximum speedup we actually measure is around 90x which implies there is scope to improve the program's distributed parallel performance. One possible improvement could include parallel file writing, negating the requirement to synchronise each thread at the end of execution. The largest contribution to the program's overall speedup was the introduction of the halo exchange reducing communication latency. Rectifying workload imbalance improved the average simultaneous CPU utilization. Interleaving communication with processing yielded some increase to speedup, however this increase is limited to the total communication latency.