HPC Report

In this report I discuss how I have successfully sped up my serially optimised Lattice Boltsman program using MPI to distribute work across 4 nodes of Blue Crystal Phase 4. I lay out the approach taken, my reasoning for this methodology, and analyse my performance considering both strong and weak scaling.

Halo Exchange

To take advantage of the parallelism that MPI provides us we need to divide our grid into smaller chunks. To calculate the value of a cell I need to read from the surrounding cells. This presents a problem for the cells on the edge of a chunk as I do not have this data. To address this I perform a Halo exchange. This is where chunks will send the necessary data to a neighbouring chunk who place this in a region called a Halo region.

In my implementation I decompose my grid by rows. This allows me to avoid unnecessarily packing the data into temporary arrays before they are sent . This is because MPI requires contiguous arrays.

I decompose my grid using the following formula.

$$\begin{aligned} work &= (rank+1)\cdot\lfloor ny/nprocs\rfloor \\ &\quad - \min(rank+1, ny\%nprocs) \\ &\quad +(rank)\cdot\lfloor ny/nprocs\rfloor \\ &\quad - \min(rank, ny\%nprocs) \end{aligned}$$

Where ny is params.ny, nprocs in the number of processors and rank is the current rank.

By using this method I ensure that the load is as balanced as possible, with the variation being at most plus or minus one row. If the load was unbalanced, for example by packing the additional rows into the final chunk, the performance can be severely degraded as the final chunk could have to compute the number of processors minus one remaining rows.

The communication strategy in my implementation was that all chunks first send to their left neighbour then sending to the right. Ranks 0 and number of processors -1 wrapped around to each other.

After the compute stage of the program, I collate the data back from the MPI rank to rank 0. There are two stages to this collating, the final state values and the av velocities.

The final state values are collated using the `MPI_Gatherv` collective. This is a variant of the `MPI_Gather` collective that sends data back to rank 0 so that it can be written to an output file. `MPI_Gatherv` varys from MPI_Gather as it allows the sending of different sizes of local domain, which happens when the domain does not equally divide amongst the processes.

The av velocities are calculated using the `MPI_Reduce` collective which allows me to sum the tot_u and tot_cells values necessary to calculate av velocities. Throughout this assignment I profiled my code using the TAU Profiler Version 2.28.2. During an early version of the code the TAU profiler revealed that approximately 30% of runtime was being spent completing MPI_Reductions. I was able to remove this by storing a copy of local `tot_u` and `tot_cells` and only completing the reduction after the compute stage.

Exploring non-blocking point-to-point API calls

MPI has non-blocking equivalents for the point-to-point communication functions. These functions will return immediately and have no guarantee that the message has been sent or buffered. The advantage of using these functions is that communication can be overlapped with computation.

Connor Hamilton - na19530 -1922777

While experimenting with these functions, I have split the for loop that operates on the local domain, so that the edge cells which read from the ghost cells are performed last. A `MPI_Waitall()` function is then used to ensure the completion of the communication. This allows as much of the calculations to happen as possible before it arrives at the wait.

Throughout these experiments consistent nodes were used, as well as multiple repeats were performed to ensure comparable results.

These experiments found that performance of using the non-blocking communication functions produces near identical performance to using the `sendrecv()` function. While some of the calculation is able to be overlapped with communication, a large proportion(~27% for the 1024x1024 grid) of runtime is being spent at the wait functions according to a TAU profiling report. While researching why the overlapping did not produce greater benefit I found that the Intel MPI library `sendrecv()` function is implemented using these non-blocking functions.

Amdahl's and Gustafson's Laws

Amdahl's Law states that the maximum possible speedup of an entire program is limited by the proportion of the program that still runs serially. The formula for calculating speed up is described below.

$$S_A = \frac{s + p}{s + \frac{p}{nprocs}}$$

Where s is the serial proportion of the program and p is the parallel proportion of the program.

I have timed the serial sections of my code which resulted in an average time of 0.13s for the grid size 128x128. This gives an estimate of 0.61% serial code. The above formula gives an estimate for the maximum possible speed up of 163 times on an infinite number of processors, and 66 times on the 112 processors available for this assignment. To gather these values I ran repeated runs using only one processor. I also timed the timing code and found this to be negligible.

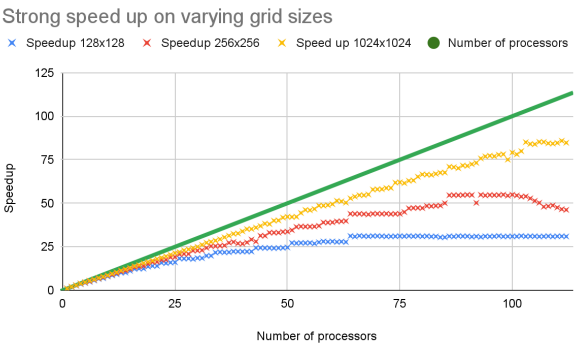
There are a number of limitations to Amdahl's Law. Firstly, there are cases where the limits set by Amdahl's Law can be exceeded. A common example of this is where the decrease in size of the local domain caused by the increased number of processors allows the problem to fit into a lower level of cache. This results in super linear speedup. Secondly, calculating the true maximum relies on the assumption of the impossible infinite number of processors. Even if every cell had its own processor, the system would also be limited by communication necessary to complete this. Thirdly, Amdahl's Law assumes that the performance of all processors is equal, which is often not the case. Finally, Amdahl's Law is constrained to work only on a fixed problem size.

Related to Amdahl's Law is the concept of strong scaling. Strong scaling refers to the speedup that occurs as processor count increases on a fixed size problem. Strong scaling speedup can be calculated with the following formula.

$$S_S = \frac{T_1}{T_{nprocs}}$$

Where T denotes the runtime on a varying number of processors.

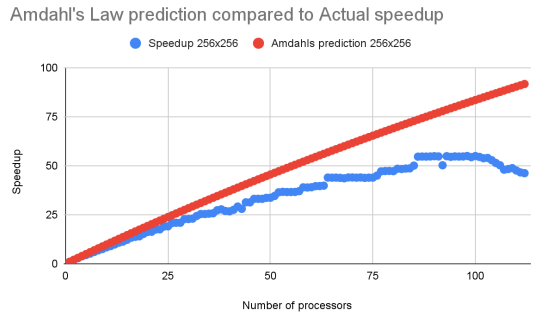
The below graph shows the scaling achieved on various problem sizes.



The above graph shows the speedups that are achieved as the number of processors increases. The green plot

shows perfect scaling that could be achieved if doubling the number of processors resulted in halfing the runtime. As the grid size increases, the program is able to continue scaling linearly for longer before experiencing sublinear performance.

In the case of the 256x256 grid, the program displays sublinear decline. There are a number of components contributing to this decline. The first of these are the limits of the serial part of the program. The below graph shows a comparison of the prediction given by Amdahl's Law and the actual speedup achieved on the 256x256 grid.



The difference between these two trends indicates that there are other factors contributing to the decline. For example, a larger amount of data that needs to be communicated and additional setup overhead for the additional ranks. For the 256x256, once the number of processors becomes greater than 100 this additional work increases to such a degree that it outweighs benefits of increased parallelisation.

As discussed already, Amdahl's Law focuses strictly on the performance of problems with a fixed size. However, I can use Gustafson's Law to understand how my program performs when the size of the problem is increased at the same rate that the number of processors is increased. Gustafson's Law states that by increasing the problem size, scalability can be retained with respect to the number of processors, and can be expressed by the following formula.

$$S_G = \frac{s + p \cdot nprocs}{s + p}$$

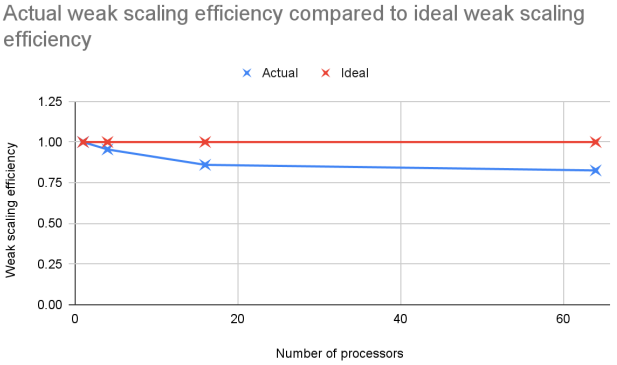
Using this formula we can estimate a maximum speedup of 111.32 times relative to a problem size that is 112 times our 128x128 grid using the full 112 processors available. Based on the same estimation of 0.61% serial work established before.

The concept of weak scaling is tightly related to Gustafson Law. Weak scaling describes how a system performs a fixed amount of work per processor as the number of processors increases. Weak scaling efficiency is described with the following formula.

$$PE_W = \frac{T_1}{T_{nprocs}}$$

Where T denotes runtime for a fixed amount of work per processor.

The graph below shows the parallel efficiency of my programs as the number of processors is increased.



From this we can see that there is some decline in weak scaling, but even at 64 processors the program is able to achieve a weak scaling efficiency of 83%.

Performance variation by node

While experimenting and testing my code, I found that depending on the nodes that ran my program the final output varied greatly. I recorded results that ranged from 7.65s to 13.41s for the 1024x1024 grid size. The ~13s results were consistent with node 100-104 and because of this I

have chosen to exclude these nodes using my Slurm batch script.

MPI Communicators

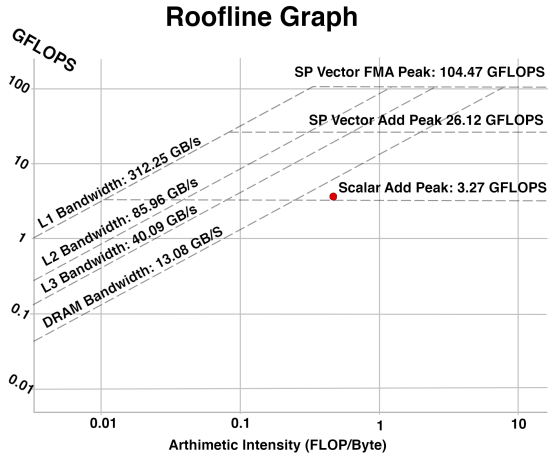
During the final testing stage of this assignment, erroneous results were identified in cases where the chunk size was equal to 1. This occurs when the number of processors is greater than half “ny” parameter. As a result of time constraints, I have restricted my program to only use at max ny/2 processors in order to always output a correct final result. This was achieved by splitting the ranks between two MPI_Communicators instead of using the default `MPI_COMM_WORLD`. The excess ranks could then be discarded and while still allowing use of MPI collectives. I have therefore excluded these values from my analysis.

Comparing OpenMP and MPI solutions

The core difference between OpenMP and MPI is that OpenMP uses shared memory and MPI uses message passing. Whilst the MPI solution was able to achieve better performance overall, it did so at a considerably greater cost. In order to match the runtime of ~14 seconds for the 1024x1024 grid my MPI required the use of 58 processors, 2.07 times as many as the OpenMP solution. A combined approach could be taken that only sends messages for Halo regions for ranks on another node, and uses the shared memory and OpenMP where possible.

Roofline analysis

A roofline graph can be produced by plotting arithmetic intensity against performance. A plot representing the peak theoretical performance of the hardware and also, a plot of the peak memory bandwidth multiplied by the operational intensity are added. The minimum of these two lines is taken and together they form the namesake roofline.



This roofline graph was generated using the 1024x1024 grid size and shows that the program is memory bandwidth bound. I have performed the STREAM benchmark which gave a peak memory bandwidth of 12.32 GFLOPS/s. Intel Advisor indicates that my program achieves 3.68 GFLOPS/s, which is 29.8% of peak memory bandwidth. This implies that there is still room to implement a more efficient solution that would decrease runtime further.

Conclusion

The best achieved runtimes for my solutions are shown in the table below as well as the run times using 112 processors.

Grid Size	128x128	256x256	1024x1024
Best times(s)	0.67	3.06	8.09
Max processor run times(s)	0.67	3.64	8.19

My results seen in the table above significantly exceed the benchmarks set out in the assignment criteria.