MPI High Performance Computing Coursework Report – Philip Mortimer [ia21629] (2100451)

Introduction

I have produced an MPI implementation that comprehensively outperforms the ballpark times for all grid sizes. I have used MPI to distribute computation over 4 BlueCrystal nodes and have explored optimisations including non-blocking MPI calls, MPI file IO, MPI collectives and using MPI in combination with OpenMP.

Simple MPI Implementation

As a first attempt, I divided the lattice Boltzmann (LBM) grid evenly between MPI ranks (ensuring that if the grid does not divide exactly between ranks that the remaining rows are evenly distributed). Before processing each timestep, the relevant halo rows are exchanged between neighbours using 6 *MPI_Sendrecv* calls to exchange the 6 required arrays (2 halo rows, with each halo row needing to send 3 cell speed arrays). Each rank then processes their section of the grid using the optimised serial code from the previous assignment.

After all timesteps are completed, the master rank collates the grid from all of the other ranks and writes it to a file. Each rank initialises the entire grid structure, though computation is only performed on the allocated elements. Running on 4 nodes with 28 cores each (4x28) on BlueCrystal, this implementation beats the ballpark times. This implementation outperforms my previous OpenMP code for larger grids as the work is being distributed over a greater number of processes.

Grid Size	*128 x 128*	*128 x 256*	*256 x 256*	*1024 x 1024*	*2048 x 2048*
Compute Time (s)	0.57	1.02	1.62	2.40	7.40

Table 1 – Processing time across all ranks of simple MPI implementation across varying LBM grid sizes (4x28).

Ensuring that each rank only stores the required section of the grid, as opposed to the entire grid, results in no noticeable speedups for compute and collation time, but offers a significant initialisation speedup. The initialisation speedup for each of the respective grid sizes is 3.0x 9.1x 19.8x 11.3x 18.7x (128 x 128, 128 x 256, 256 x 256, 1024 x 1024 and 2048 x 2048 grids). Any tuple of five speedups from this point on refers to these grid sizes in this order, using 4 node and 28 cores. The absolute initialisation time is between 0.001 and 0.004 seconds.

MPI_Isend and MPI_Irecv

MPI_Isend and *MPI_Irecv* calls return immediately and allow for processing to happen whilst halo rows are exchanged. *MPI_Sendrecv* calls within the timestep function have been replaced with these calls. This allows for each rank to update the grid for all cells that don't need to access a halo row whilst waiting for communication to finish. Once these cells have been updated, *MPI_Waitall* is used to ensure that all relevant halo rows have been received, and the remaining cells are then computed. After all computation is done, but before the next timestep, *MPI_Waitall* is called again to ensure that all send operations are finished.

This change results in a 1.3x 1.25x 1.1x 1.2x 1.2x speedup. This speedup occurs as each rank is making use of time that would've been spent waiting for ranks to communicate with each other to perform a large part of the computation. After this changes, for the 2048x2048 grid, 28% of CPU time is spent waiting for communication (in the *MPI_Waitall* and *psm2_mq_ipeek functions*) as measured using Intel VTune.

MPI File IO

Concurrent Grid Write

Currently, each rank sends their segment of the grid to the master rank after all timesteps have been computed (using 9 MPI send cells per rank – one for each speed matrix). The master rank collates the entire grid and writes it to the output file, *final_state.dat*. MPI supports parallel file IO, allowing each rank to concurrently write to different chunks of the same file. This can be utilised to avoid having to synchronise the final grid state and thus significantly reducing collate time.

Each line of the *final_state.dat* file is of the following format: *"ii jj u_x u_y u pressure obst\n"* and thus can be written to concurrently. The output file is opened with *MPI_File_open()*, using the *MPI_MODE_CREATE* and *MPI_MODE_WRONLY* flags. *MPI_File_set_size()* is called to ensure the file size is correct. After this, each rank can simply write their section of the grid to their section of the file using *MPI_File_write_at_all*.

Doing this significantly reduces the collate time, resulting in a 1.1x 1.4x 1.1x 2.7x 15.1x speedup. For larger grid sizes, the reduced collation time is substantial. It is important to note that these measurements are somewhat noisy as collation is quick, taking around 0.0071 seconds for 2048 x 2048 cells.

Reductions for Calculating Average Velocities

The other output file, *av_vels.dat*, stores the average velocity of the entire grid at each timestep. Each line in the file is of the form "*timestep average_velocity*\n". As such, the file can't be divided by rank. All the data must be sent to the master rank which then writes the velocities to file. To make synchronisation more efficient, *MPI_Reduce* is used to sum the total number of cells and cumulative velocity. These two arrays are utilised to compute the average velocity in a vectorised for loop. Using *MPI_Reduce* to collate all velocities improves on using point-to-point MPI communication with the master rank as MPI may use tree-based communication to reduce the number of messages sent. This change results in a significant collate speedup (1.5x 2.8x 2.5x 3.0x 6.5x).

MPI + OpenMP

I experimented with a solution that combines MPI and OpenMP. To enable multithreaded MPI, *MPI_Init* is replaced with *MPI_Init_thread()*, using the *MPI_THREAD_FUNNELED* flag to indicate that each MPI rank may be multi-threaded but that only the master thread will make MPI calls. The program is compiled with the *-qopenmp* and *-mt_mpi* flags which enable OpenMP and multi-threading for MPI.

The program is parallelised by applying a *#pragma omp parallel for* directive to the outer loop of the LBM computation (which iterates through grid rows). The inner loop is a vectorised loop over cells within a specific row. Both loops require a reduction over the two variables summing the total velocities and total cells. Using this, each MPI rank distributes the core LBM computation across OpenMP threads. Cell values are initialised in parallel to achieve optimal NUMA speeds.

For hybrid MPI / OpenMP programs, it is typically desirable to attach each MPI rank to a socket and then to ensure that all of the rank's OpenMP threads remain within that socket. To achieve this, *I_MPI_PIN_DOMAIN=omp*, *I_MPI_PIN=yes*, *OMP_PLACES=cores*, *OMP_PROC_BIND=close* and *OMP_NUM_THREADS= $SLURM_CPUS_PER_TASK* can be used.

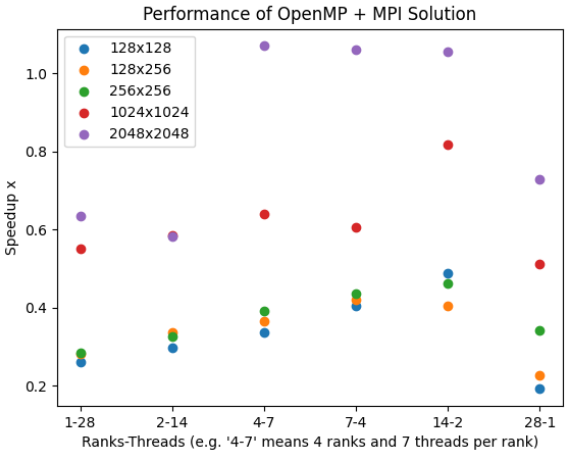

Figure 1 – Performance of OpenMP solution as number of threads vs ranks differs using 4 nodes. Speedup is given relative to performance of final MPI solution (see Table 2).

Figure 1 outlines how performance varies as the number of ranks per node changes. In most cases, optimal performance was obtained when using 14 ranks and 2 threads. This differs slightly from the expected choice of 4 ranks and 7 threads, which may suggest that the thread pinning policy does not perform as expected. It may also be a result of less effective vectorisation within parallel regions. For the 2048x2048 grid, 4 ranks and 7 threads perform best. This is also the only grid size for which the OpenMP solution outperforms the pure MPI solution, achieving a runtime of 5.55 seconds. I hypothesise that OpenMP makes the solution more scalable and that for even larger grid sizes, the speedup would be more substantial. OpenMP uses shared memory within a rank and thus reduces the number of MPI calls that need to occur. This reduces the time spent waiting for Halo communication.

Finetuning

As my pure MPI program performs best, I have elected to use this as my final submission. I use the intel compiler (*mpiicc*) along with the following compiler flags: *-O3 -xHost -ipo*. It is interesting to note that enabling the *-0fast* flag leads to the 2048 x 2048 grid average value check failing (though all other checks pass) with the biggest difference in average velocity and expected value being 2%. This happens as *-0fast* enables unsafe floating point maths operations which lead to this small error. The performance difference between *-03* and *-0fast* is negligible, so this is not an issue.

Scaling

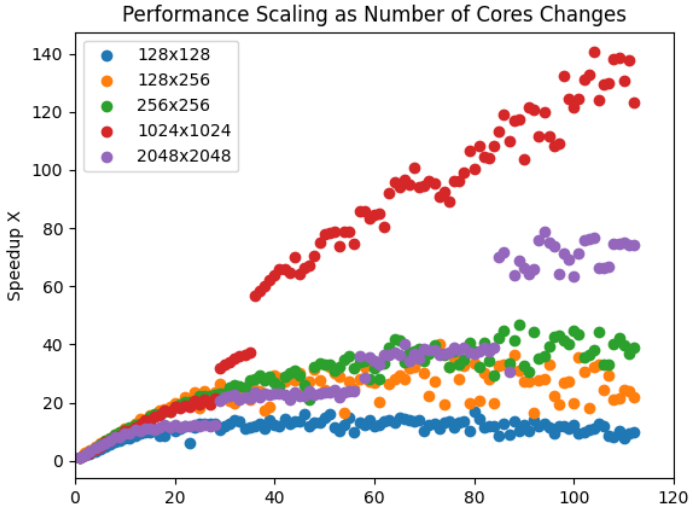

Figure 2 – Performance of pure MPI solution as number of cores changes. Speedup is given relative to final performance of pure MPI solution for each grid size (see Table 2).

Figure 2 outlines performance scaling of the pure MPI solution as the number of cores vary. Increasing the number of cores improves performance. For smaller grid sizes this trend continues until a plateau is reached (sublinear plateau), where adding additional cores leads to no speedup and can even degrade performance. Initially, speedup occurs as work is being parallelised. For smaller grids, exceeding 28 cores leads to a decrease in the rate of speedup as ranks must communicate between nodes, which is more expensive than intra-node communication.

For the 1024x1024 grid, linear scaling occurs, with the number of cores almost perfectly corresponding to speedup. There is a dramatic speedup when going from 28 to 29 cores (when an additional node is added). This speedup occurs as adding an additional node increases the total L3 cache available. For the 2048x2048 grid, a sublinear plateau occurs within each node until a new one is added. This problem size does not scale as

effectively as 1024x1024, as the larger grid size leads to more cache misses. The 2048x2048 grid has a 25.1% cache miss rate, compared to just 1.2% for 1028x1028 (measured using Linux's *prof* utility).

Roofline Analysis

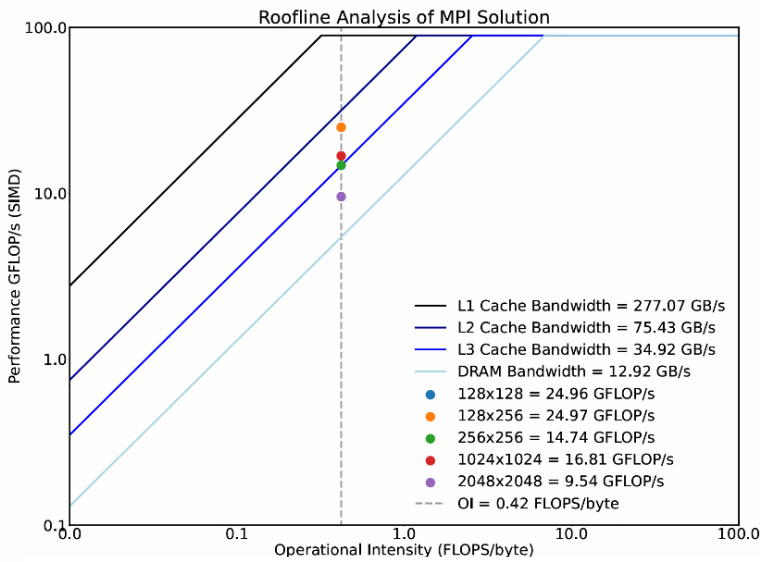

Figure 3 – Roofline analysis of MPI solution. Note that 128x128 and 128x256 have near identical performance and hence occlude each other. Roofline is based on 4 nodes each with 28 cores (with roofline showing average performance per rank).

Based on the roofline analysis in *Figure 3*, the program is clearly memory bandwidth bound. The problem sizes sit between DRAM and L2 cache and is (likely) bound by L3 cache for larger problem sizes. For smaller problem sizes, it is (likely) bound by L2 cache or even L1 cache. This is because each rank only processes a sufficiently small portion of the grid that can be entirely stored in quick access cache. For instance, for the 128x128 grid, when using all 112 cores, each core only stores 12 KB of data. This is smaller than the 32 KB of L1 cache each core has.

The LBM calculation achieves 64% of peak STREAM DRAM bandwidth (mean performance of triad, add, copy and scale STREAM benchmarks), indicating that the code is well optimised. This is calculated based on the DRAM bandwidth of 12.9 GB/s per core compared to the amount of data processed throughout runtime on the 1024x1024 grid using:

$$perf\left(\frac{B}{s}\right) = \frac{nx*ny*2*9*size(float)*iterations}{runtime*cores}$$ giving a bandwidth of $\frac{1024*1024*2*9*4*20000}{1.63*112} \cong 8.3GB/s$

Final Performance

Grid Size	*128 x 128*	*128 x 256*	*256 x 256*	*1024 x 1024*	*2048 x 2048*
Initialisation Time (ms)	1.1	0.9	1.0	1.8	3.3
Compute Time (s)	0.37	0.44	0.90	1.63	6.20
Collate Time (ms)	1.8	15.3	4.0	2.4	2.1
Total Time (s)	0.37	0.45	0.91	1.63	6.21

Table 2 – Runtime of final submission. Figures are fastest total time over 20 runs for each grid size.

My MPI solution outperforms my OpenMP code (which only uses one node) from the previous assignment for all problem sizes, with a relative speedup of 1.4x 1.8x 2.9x 7.1x 7.8x. It is also 3.0x 4.5x 5.5x 2.9x faster than the provided ballpark times (note that a ballpark time for the 128x256 grid has not been provided).

Conclusion

I have produced a fast, scalable MPI solution that outperforms all benchmark times by quite some margin. I have explored MPI IO, non-blocking MPI calls, MPI collectives and combining OpenMP with MPI to obtain my final LBM solution.