Parallelising the Lattice Boltzmann Method

George Herbert
cj19328@bristol.ac.uk

27 April 2022

Abstract

The Lattice Boltzmann method (LBM) simulates a fluid density on a lattice. This report analyses the techniques I utilised to produce three implementations of LBM: one with distributed-memory parallelism, using Message Passing Interface (MPI) for interprocess communication; one hybrid, using a combination of MPI and OpenMP; the third used OpenCL to run on a GPU.

1 Single Program, Multiple Data

1.1 Hypothesis

My OpenMP implementation of `d2q9-bgk.c` with shared-memory parallelism achieved a substantial speedup over my optimised serial implementation. However, it was limited to a single node of BC4, which was a considerable restriction.

Single program, multiple data (SPMD) is a form of parallelism in which independent processes run the same program. Message Passing Interface (MPI) is a specification for a library interface for passing messages between processes. Therefore, I hypothesised that implementing of `d2q9-bgk.c` with distributed memory parallelism that ran on multiple processes across multiple nodes—with MPI used for interprocess communication—would provide an even more significant speedup.

1.2 Compiler

I used the `mpiicc` wrapper script, which compiled my program with the Intel C Compiler Classic (version 19.1.3.304) and set up the include and library paths for the Intel MPI library. I compiled my program with the `-std=c99`, `-Wall`, `-Ofast`, `-restrict` and `-xAVX2` options.

1.3 Load Balancing

I had to assign different sections of the grid to different processes explicitly. The distribution needed to be adequately balanced to minimise the time processes' spent blocked.

Since the `cells` grid was in row-major order, I split the grid horizontally between processes to take advantage of memory locality. I created a procedure `allocate_rows` to balance the load. The procedure assigned each process at least $\lfloor \frac{y}{n} \rfloor$ consecutive rows, with the first $y - \lfloor \frac{y}{n} \rfloor n$ processes each assigned an additional consecutive row, where y was the number of rows and n the number of processes. Additionally, since updating the value of a given cell required the values of all adjacent cells, each process contained two additional rows reserved for cells in the top and bottom rows of the preceding and succeeding ranks, respectively. Figure 1 displays an example allocation for a grid with five rows split between two processes; the green rows are those allocated to the process, whilst the red rows are the additional rows required to update the edge rows correctly.

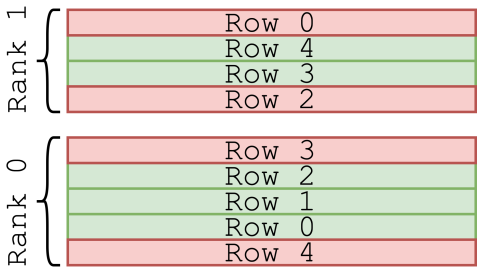

Figure 1: Row allocation example with five rows and two processes

I decided against splitting on a sub-row level to avoid unnecessarily increasing the complexity of my program and incurring additional computational overhead.

1.4 Halo Exchange

Since processes are assigned an individual virtual memory region, I had to explicitly send the contents of edge rows to neighbouring ranks after each timestep. To do so, I created a `halo_exchange` procedure. The procedure copied the bottom-most row allocated to the process into the `send_row_buffer` array. Since the `cells` data was in the structure of arrays (SoA) format, the values from the nine separate arrays of speeds had to be packed into the buffer. I used the `MPI_Sendrecv` procedure to send this buffer to the `receive_row_buffer` of the preceding rank. Then, the procedure copied the values into the additional upper row—unpacking the values into the nine separate arrays of speeds. The same process was then repeated for the top-most row, which was sent to the succeeding rank.

1.5 Collating

I created a `collate` procedure that was executed once all iterations of the `timestep` procedure were complete.

The procedure had two purposes. The first purpose was to transfer the final state of the cells allocated to each process to the master process (i.e. rank zero). The second purpose was to transfer the partial average velocity values to the master process and use these values to calculate the correct average velocity at each timestep.

I used the `MPI_Send` procedure to send the final state of the cells allocated to each process to the master process. The master process received these values by executing the `MPI_Recv` procedure once for each process.

I also used the `MPI_Send` procedure to send each process' partial average velocity values to the master process. The master process received these arrays of values consecutively from each process using the `MPI_Recv` procedure and summed them into a global average velocities array. The `collate` procedure then multiplied each element by `params.num_non_obstacles_r` in the master process to calculate the correct average velocity at each timestep.

1.6 Results

Table 1 displays the results of my MPI implementation. Each time was an average of three runs on BlueCrystal Phase 4 (BC4) compute nodes, which were each a Lenovo nx360 M5, which contained two 14-core 2.4 GHz Intel E5-2680 v4 (Broadwell) CPUs and 128 GiB of RAM [1]. My MPI implementation provided a substantial speedup over both my optimised implementation and my OpenMP implementation. My program achieved the speedup by executing sections of the `timestep` procedure in parallel, utilising 112 times more L1 and L2 cache and utilising eight times more L3 cache.

Table 1: Execution times with the 112 process MPI implementation and speedup over both my optimised serial and 28 thread OpenMP implementation

		Speedup	
Grid Size	Time (s)	Serial	OpenMP
128×128			
128×256			
256×256			
1024×1024			

2 Optimisations

2.1 Average Velocities Reduction

I hypothesised I could use the `MPI_Reduce` procedure to speed up the collation time of my program. Table 2 contains the collation times for this implementation and speedup over the prior implementation. Using `MPI_Reduce` led to a significant reduction in collation time. However, this did not significantly affect the overall execution time because the collation time was short compared to the compute time. I would expect a more significant impact on the overall execution time for inputs containing more iterations.

Table 2: Collate times with the reduction and speedup over the prior implementation

Grid Size	Time (s)	Speedup
128×128	`0.0016`	`4.54`
128×256	`0.0029`	`2.93`
256×256	`0.0049`	`2.83`
1024×1024	`0.0420`	`1.08`

3 Analysis

3.1 Comparison to Optimised Serial

My MPI implementation achieved a substantial speedup over my optimised serial implementation. To identify the merits and demerits of both implementations, I used the Intel Advisor tool to analyse their performances.

as shown in Figure 2.

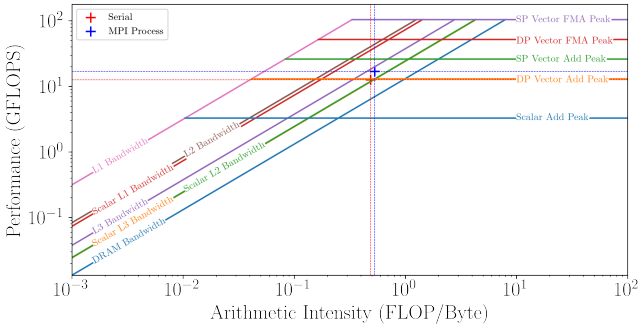

Figure 2: Roofline model of the `timestep` loop run on the 1024×1024 test case

On the 1024×1024 test case, each process in my MPI implementation achieved an arithmetic intensity of 0.533 FLOP/Byte and 16.905 FLOP/s of performance during the `timestep` procedure. Compared to my optimised serial implementation, the arithmetic intensity was approximately identical, which was expected since I made very few changes to the procedure. However, the performance increased by 35% because the program did not have to interact with the DRAM, which had a lower bandwidth, as often. For example, in the `timestep` procedure in my MPI implementation, less than 0.01 GB of data was passed through DRAM. In comparison, in my optimised serial implementation, 2306.03 GB of data was passed through the DRAM.

My MPI implementation had a large overhead introduced by the `halo_exchange` procedure. In fact, when ran with 112 ranks on the 1024×1024 test case, 23% of the compute time of my program was spent in this procedure.

3.2 Scaling

I ran my final MPI implementation from 1–112 processes to analyse how my program scaled. My program ran on as few nodes as possible, with each process assigned to a single core. I then calculated the subsequent processes'

speedup over a single process implementation. Figure 3 displays the result speedup curves.

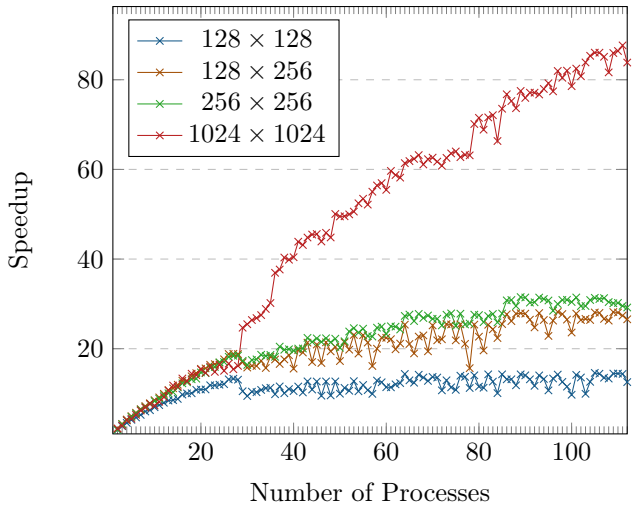

Figure 3: Speedup curves for my MPI implementation

In general, my implementation initially scaled well for each grid size, but the speedup aquired from each subsequent process declined (i.e. a sublinear plateau). Perfect linear scaling was impossible to achieve for several reasons. Firstly, the program contained serial sections, notably the `initialisation` and `collate procedures`; in fact, the duration of the `collate` procedure increased with each additional process. Secondly, since the number of processes was rarely a factor of the y length of the grid, there was often a small amount of load imbalance.

Notably, larger grid sizes benefitted more from a distributed parallel implementation than the smaller grid sizes. Firstly, this was because the larger grids benefitted more from being split sufficiently small to fit into the smaller cache levels with higher memory bandwidths. Secondly, the larger grid sizes were more evenly divided by the number of threads.

There were some deviations from a pure sublinear plateau. For the 128×128 grid there was a small decline in speedup after 28 processes, which was due to the latency of the InfiniBand connection between nodes. In contrast, for the 1024×1024 grid, there was a clear increase in speedup after approximately 30 processes. As mentioned, the superlinear speedup at this point was caused by the largest problem size being split sufficiently small to fit into the L3 cache.

4 Hybrid MPI and OpenMP

I sought to investigate whether a hybrid implementation that used both MPI and OpenMP would execute faster than my prior implementation.

4.1 OpenMP vs MPI Scaling

Before producing my hybrid implementation, I compared the speedup curves of my MPI implementation to those of my OpenMP implementation. Figure 4 displays the speedup curves of these two implementations for the 128×128 and the 1024×1024 grid sizes.

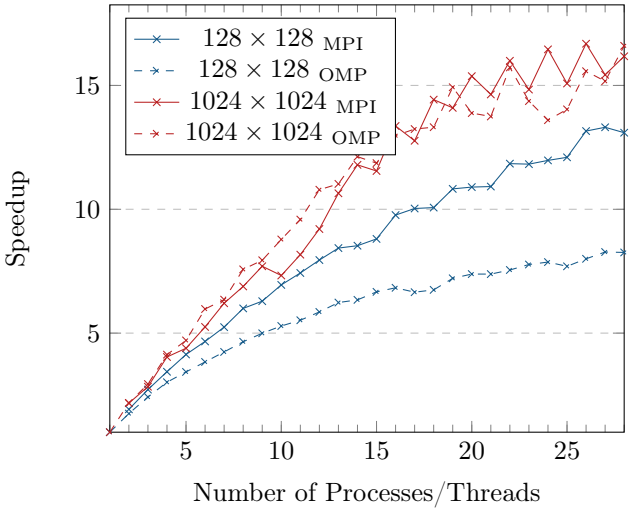

Figure 4: Speedup curves for my OpenMP and MPI implementation

My MPI implementation scaled better for the smaller grid, whereas my OpenMP implementation initially scaled slightly better for the larger grid. This was because each implementation had a different overhead: in the OpenMP implementation, the overhead arose from the creation and synchronisation of threads each timestep; in the MPI implementation, the overhead arose from the halo exchange. The OpenMP overhead was largely constant regardless of the grid size, whereas the MPI overhead was correlated with the width of the grid.

4.2 Implementation

Having gained an understanding of how my OpenMP and MPI implementations scaled, I hypothesised that a hybrid implementations would reduce the execution times for large, wide grid sizes. Therefore, I produced three additional grids to test my implementation with.

Using my prior implementation as a starting point, I replaced the call to the `MPI_Init` procedure with a call to the `MPI_Init_thread` procedure, passing `MPI_THREAD_FUNNELED` as the third argument since only the main thread was to make MPI calls. I parallelised the outer loop in the `timestep` procedure with OpenMP, and set the `I_MPI_PIN_DOMAIN` environment variable to `socket`.

I tested my implementation with eight processes—one per socket across four nodes—communicating via MPI, with each process creating fourteen threads. Table 3 displays the results of my experiment. As anticipated, the hybrid implementation was slower with the smaller grids that were provided, but faster with the larger grids that I produced myself.

5 GPU Programming

OpenCL is a framework for heterogeneous computing that can be used for GPU programming. GPUs typically have 3–5x the memory bandwidth, and 5–10x the peak FLOP/s that CPUs have. This is true for BC4, in which a single NVIDIA Tesla P100 has 4.8x the peak memory

Table 3: Execution times with the hybrid implementation and speedup over the prior implementation

Grid Size	Time (s)	Speedup
128×128	0.67	
128×256	0.73	
256×256	1.94	
1024×1024	3.17	
4096×1024	15.99	1.42
8192×1024	53.88	1.03
16384×1024	102.34	1.15

bandwidth, and 9.8x the peak double precision performance that two Intel E5-2840 v4 CPUs have [2]. Therefore, I sought to produce an implementation of LBM built with OpenCL to run on a GPU.

5.1 Implementation

I used my optimised serial implementation as a starting point for my host program. To ensure my program was portable, I used the `clGetPlatformIDs` and `clGetDeviceIDs` procedures to produce an array of all devices available to the host program, and selected the device defined by the `OCL_DEVICE` environment variable. For the selected device, I created a single context and a single in-order queue. I kept `cells` and `cells_new` data in the SoA format, since coalesced memory accesses were key for high bandwidth.

I converted the `accelerate_flow` and `timestep` procedures into kernels to run on the GPU. Transerring memory between host and device is a slow operation; therefore, I opted to store the partial average velocities of each timestep in global memory on the device. Each timestep, the `timestep` kernel performed a parallel reduction to sum the velocities of each cell in the same work group. I opted to implement a parallel reduction to minimise the number of addition operations. I summed the average velocities of each work group on the host device once all iterations had been complete.

Table 4 displays execution times of my OpenCL implementation when run on a single NVIDIA Tesla P100, and speedup over my MPI implementation when ran with 28 processes on a single node of BC4. The OpenCL implementation was slower for the smaller grid sizes, but faster for the larger grid sizes. One reason for this was because for the smaller grids, the time spent by the host program to set up and manage the environment, and to create and manage the kernels represented a greater proportion of the runtime. Another reason was because the smaller grids fitted into the smaller cache levels with higher memory bandwidths on my MPI implementation.

5.2 Work-Group Size

In OpenCL, kernels execute in parallel over a predefined N-dimensional domain, whereby independent element of execution within this domain is known as a work-item. These work-items are often grouped together into independent work-groups. I previously used a

Table 4: Execution times with the OpenCL implementation and speedup over my optimised serial implementation

Grid Size	Time (s)	Speedup
128×128		
128×256		
256×256		
1024×1024		

I previously used a local size of 16 cells in the y direction, and 32 cells in the x direction.

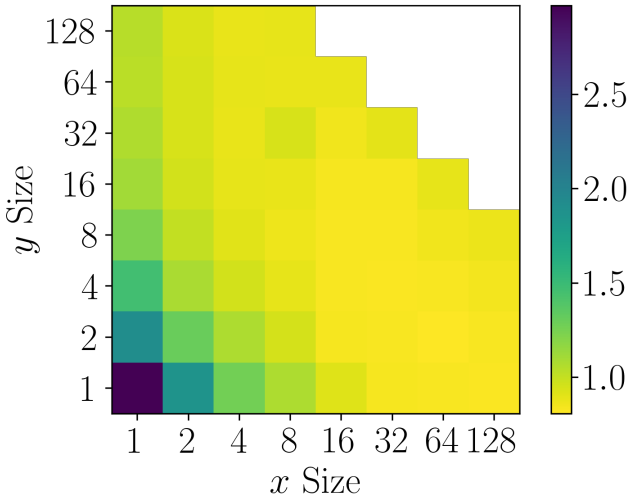

Figure 5: Compute times for the 128×128 grid ran with different local sizes

6 Conclusion

In conclusion, all three of the implementations I produced and tested significantly improved the performance of `d2q9-bgk.c`.

References

[1] *BlueCrystal technical specifications.* URL: `https://www.bristol.ac.uk/acrc/high-performance-computing/hpc-systems-tech-specs/` (visited on 19/02/2022).

[2] Simon McIntosh-Smith. *HPC week 8 lecture 1 GPU programming intro.* URL: `https://www.youtube.com/watch?v=CuIdzH90Lwk` (visited on 17/04/2022).