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 implementation `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. The first $y - \lfloor \frac{y}{n} \rfloor n$ processes were assigned an additional consecutive row, where y was the number of rows and n was 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.

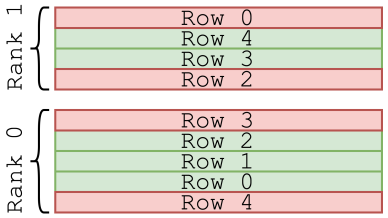

Figure 1: Row allocation example with five rows and two processes

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 were 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 procedure then repeated the same process for the top-most row, sending it to the next 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 pro-

cess. 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 my optimised serial implementation and my OpenMP implementation. My program achieved the speedup over my optimised serial implementation by executing sections of the `timestep` procedure in parallel, utilising 112 times more L1 and L2 cache and 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

Grid Size	Time (s)	Speedup	
		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 significantly shorter the compute time. I would expect a more significant impact on the overall execution time for inputs containing more iterations.

3 Analysis

3.1 Comparison to Optimised Serial

My 128 process MPI implementation achieved a substantial speedup over my optimised serial implementation.

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`

To identify the merits and demerits of both implementations, I used the Intel Advisor tool to analyse their performances. Figure 2 displays a roofline chart for the `timestep` procedure of both implementations.

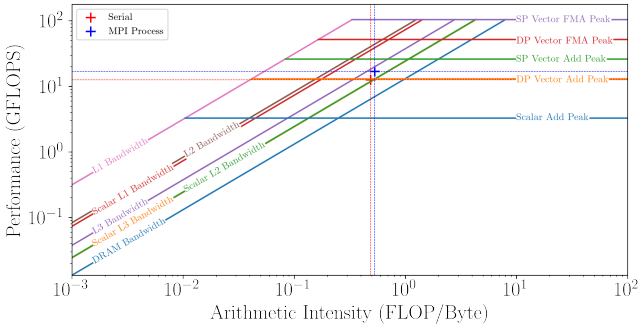

Figure 2: Roofline model of the `timestep` loop run on the 1024×1024 test case

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. The arithmetic intensity was similar to my optimised serial implementation because I made only minor changes to the procedure. However, the performance increased by 35% due to fewer interactions with the DRAM, which had a lower memory bandwidth. For example, the `timestep` procedure in my MPI implementation, less than 0.01 GB of data was passed through DRAM. In comparison 2306.03 GB of data was passed through the DRAM in my optimised serial implementation.

My MPI implementation had a significant overhead introduced by the `halo_exchange` procedure. When executed for the 1024×1024 test case with 112 processes, my program spent 23% of the compute time 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.

In general, my implementation initially scaled well for each grid size, but the speedup acquired 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,

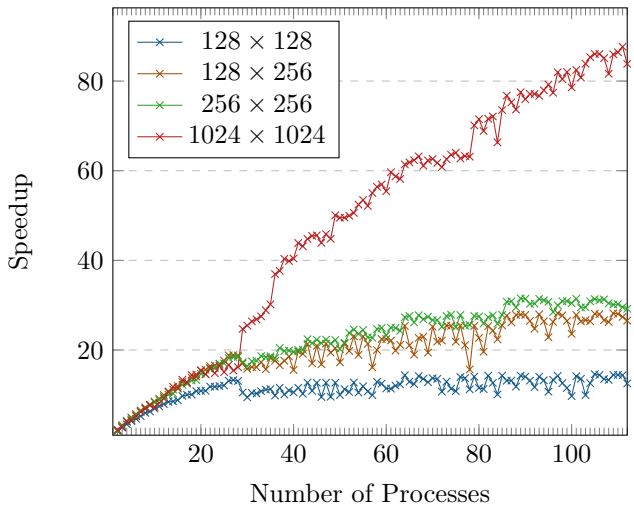

Figure 3: Speedup curves for my MPI implementation

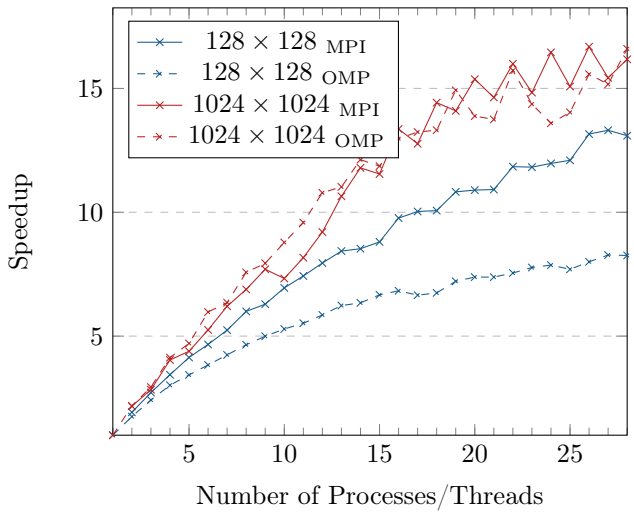

Figure 4: Speedup curves for my OpenMP and MPI implementation

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 grid's y-length, there was often a tiny load imbalance.

Notably, larger grid sizes benefitted more from a distributed parallel implementation than the smaller ones, which occurred for several reasons. Firstly, 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, a slight decline occurred after 28 processes due to the latency of the InfiniBand connection between nodes. In contrast, for the 1024×1024 grid, there was an apparent increase in the rate of speedup after approximately 30 processes. As mentioned, the superlinear speedup at this point occurred because the grid was split sufficiently small to fit into the L3 cache.

4 Hybrid MPI and OpenMP

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

My MPI implementation scaled better for the smaller grid, whereas my OpenMP implementation initially scaled slightly better for the larger grid. The scaling differences occurred 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 essentially constant regardless of the grid size, whereas the MPI overhead was correlated with the grid's width.

4.2 Implementation

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

Using my prior implementation as a starting point, I replaced 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 provided but faster with my produced grids.

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

5 GPU Programming

OpenCL is a framework for heterogeneous computing used for GPU programming. GPUs typically have 3–5x the memory bandwidth and 5–10x the peak FLOP/s that CPUs have. In BC4 a single NVIDIA Tesla P100 has 4.8x the peak memory 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. I selected the device defined by the `OCL_DEVICE` environment variable and created a single context and a single in-order queue for the device. 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. Transferring 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. For 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 my OpenCL implementation's execution times when run on a single BC4 GPU node, and speedup over my 28 process MPI implementation run on a single compute node. The OpenCL implementation was slower for the smaller grid sizes but faster for the largest grid sizes. One reason for this was because for the smaller grids, the time taken by the host program to create and build the program, set up and manage the environment, and create and manage the kernels represented a far more significant proportion of the runtime. For example, for the 128×128 grid, creating and building the program constituted almost 60% of the total execution time of my OpenCL program. Another reason was that the smaller grids fitted into the smaller cache levels with higher memory bandwidths on my MPI implementation.

Table 4: Execution times with the OpenCL implementation and speedup over my 28 process MPI implementation

Grid Size	Time (s)	Speedup
128×128	`1.85`	`0.21`
128×256	`1.83`	`0.34`
256×256	`2.69`	`0.84`
1024×1024	`4.81`	`2.83`

5.2 Work-Group Size

In OpenCL, kernels execute in parallel over a predefined N-dimensional domain, whereby independent execution elements within this domain are known as work-items. These work-items are often grouped into independent work-groups; within a work-group, synchronisation between work-items is possible. Work-group size can drastically impact performance; fewer work-groups reduce the overhead of maintaining the work-groups, but larger work-groups can be slower if the kernel code contains the barrier instruction. To find the optimal work-group size for the 128×128 grid, I ran my OpenCL program using a variety of work-group sizes, as shown in Figure 5. The optimal local size was 128 cells in the x-direction, and one cell in the y-direction, therefore using 128 work-groups.

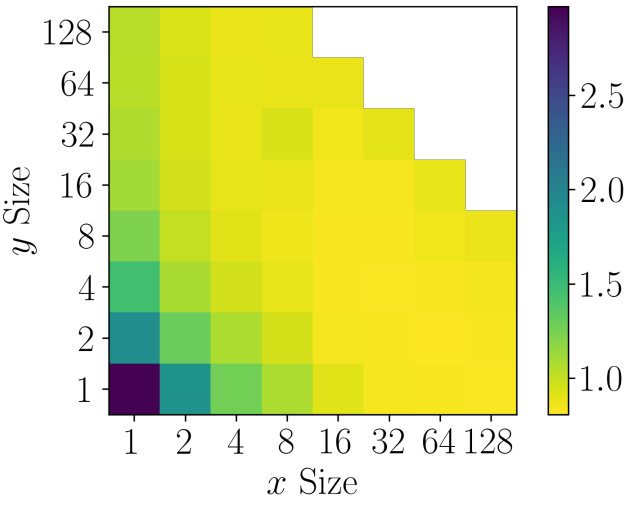

Figure 5: Compute times for the 128×128 grid ran with different local sizes

6 Conclusion

In conclusion, all three of the implementations I produced and tested significantly improved the performance of `d2q9-bgk.c` by utilising different levels of parallelism. For the largest grid size, my OpenCL implementation was the fastest on a single node due to GPUs' large memory bandwidth and peak performance. However, both my MPI and hybrid implementations were more scalable since they could utilise the computing power of multiple nodes.

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).