Parallelising d2q9-bgk.c with MPI

George Herbert
cj19328@bristol.ac.uk

22 April 2022

Abstract

d2q9-bgk.c implements the Lattice Boltzmann method (LBM) to simulate a fluid density on a lattice. This report analyses the techniques I utilised to parallelise d2q9-bgk.c with MPI, and port d2q9-bgk.c to a GPU with OpenCL.

1 Single Program, Multiple Data

1.1 Hypothesis

I previously achieved a substantial performance improvement producing a multithreaded implementation of d2q9-bgk.c with OpenMP. However, since OpenMP was built for shared-memory parallelism, my implementation could not utilise more than one 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 hypothessied that a parallel implementation of d2q9-bgk.c that ran on multiple processes across multiple nodes—with MPI being used for interprocess communication—would provide an even more substantial performance improvement.

I opted to use the implementation of my program prior to enforcing single instruction, multiple data (SIMD) vectorization as a starting point. I was uncertain the changes I previously implemented to enforce SIMD vectorization would provide a performance benefit to my MPI implementation.

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.

1.3 Load Balancing

I had to explicitly assign different sections of the grid to different processes. It was crucial for the distribution to be adequately balanced to minimise the amount of time processes' spent blocked.

Since the cells grid was stored 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 rows allocated to a specific process are highlighted in green, with additional rows required to correctly updated the edge rows highlighted in red.

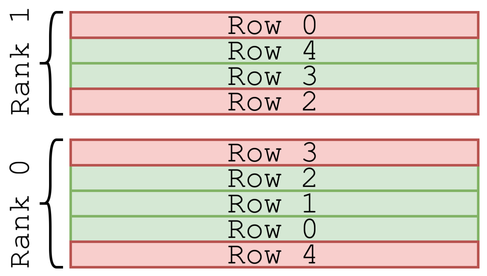

Figure 1: Row allocation example with five rows and two processes

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

1.4 Halo Exchange

Since processes are assigned their own virtual memory region, I had to explicitly send the contents of edge rows to neighbouring ranks at the conclusion of each timestep. To do so, I created a halo_exchange procedure. The procedure copied the bottom-most row allocated to each process into the send_row_buffer array. I used the MPI_Sendrecv procedure to send this buffer to the receive_row_buffer of the preceding rank. The values in the receive_row_buffer were then copied into the top additional row. 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 to be executed once all iterations of the timestep procedure were complete. The procedure had two purposes. The first purpose was to transfer the 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 the partial average velocity values held by each process 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. Once the arrays had been summed, the colate procedure 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 prior implementation (before enforcing SIMD vectorization), as well as my OpenMP implementation. The speedup is primarily achieved by utilising additional cores to execute sections of the timestep procedure in parallel.

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

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

2 Optimisations

2.1 Vectorization

I hypothesised that SIMD vectorization of the inner loop would drastically improve the performance of my MPI implementation, as it did with my serial optimised implementation previously. Therefore, I made the same changes as I did with my serial optimised implementation, including converting the cells' data from an array of structures (AoS) to a structure of arrays (SoA) format. Implementing the SoA format meant that the halo_exchange procedure had to be altered since the MPI_Sendrecv procedure required the address of a single buffer as input.

I could have implemented nine separate MPI_Sendrecv procedures—one for each speed. However, I instead decided to copy each of the nine speeds into a single array, followed by a single call to the MPI_Sendrecv. I selected this route, since each call to the MPI_Sendrecv procedure has an associated overhead. However, this is why I did not enforce SIMD vectorization in my prior implementation, since I could not be certain the performance benefit gained would outwigh the drawback to performance that arises from copying the nine speeds into a single array.

Table 2 displays the execution times and speedup after enforcing SIMD vectorization. Enforcing SIMD vectorization provided a substantial speedup.

Table 2: Execution times after enforcing SIMD vectorization and speedup over the prior implementation

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

2.2 Average Velocities Reduction

I hypothesised I could use the MPI_Reduce procedure to reduce the collation time of my program. Table 3 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 make a large difference to the overall execution time due to the collation time being so short compared to the compute time. I would expect a more significant impact on the overall exeuction time for inputs containing more iterations.

Table 3: 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 Experiments

3.1 Hybrid MPI and OpenMP

I sought to investigate whether a hybrid implementation would execute faster than my prior implementation. Therefore, I produced an implementation of d2q9-bgk.c that ran with eight processes—one per socket across four nodes—communicating via MPI, with each process branching fourteen threads at specific points in the program. These specific points were identical to that in my OpenMP implementation: the outer loop in the timestep procedure, and the initialisation loops for the obstacles and cells arrays.

I tested my program with the four grids provided, as well as three additional grids that I produced. Table 4 displays the results of my experiment. The hybrid implementation was slower with the smaller grids that were provided, but faster with the larger grids that I produced myself. This was because, with the smaller grids, the overhead introduced by creating and synchronising OpenMP threads each timestep outweighed the performance benefit of fewer halo exchanges. However, the hybrid implementation excelled with the larger grids, since the overhead introduced by OpenMP was near identical, but the time spent performing halo exchanges increased dramatically.

Table 4: Execution times with the hybrid implementation and speedup over the prior implementation

Grid Size	Time (s)	Speedup
128 × 128	0.66	
128 × 256	0.73	
256 × 256	1.93	
1024 × 1024	3.19	
4096 × 1024	17.18	1.32
8192 × 2048	91.74	1.04
16384 × 4096	337.16	1.01

3.2 OpenMP vs MPI

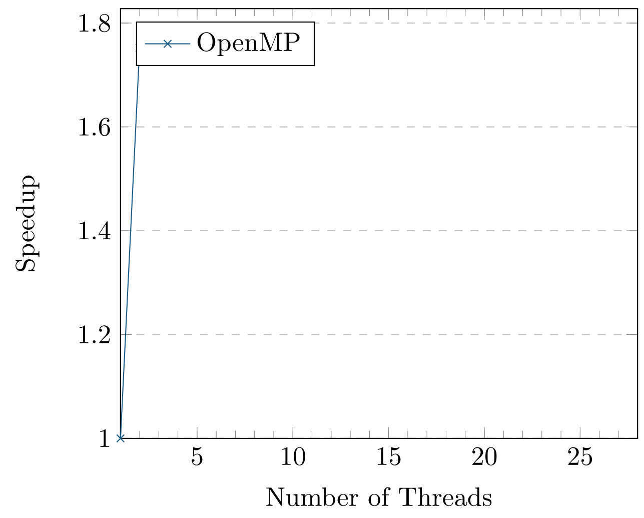

Figure 2: Speedup curves for my OpenMP and MPI implementation on the 1024 × 1024 grid

3.3 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, such that each process was 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 the smallest three grid sizes, but the speedup acquired from each subsequent process declined (i.e. a sublinear plateau), which occurred due to several reasons. Perfect

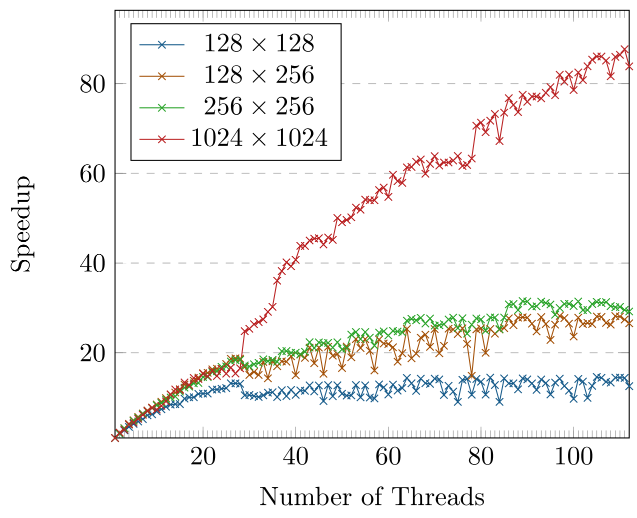

Figure 3: Speedup curves for my MPI implementation

linear scaling was theoretically impossible because the program contained serial sections.

A different pattern emerged in the largest grid size.

Notably, the amount of speedup provided by each next core was approximately inversely proportional to the test case size. In other words, larger grid sizes benefitted more from a multiprocess implementation than smaller grid sizes. Firstly, this was because the larger grids benefitted more from being split sufficiently small to fit into the faster cache levels. Secondly, the larger grid sizes were more evenly divided by the number of processes.

4 Comparison to Serial

5 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 the Nvidia Pascal P100 has 4.8x the memory bandwidth and 9.8x the performance that the Intel E5-2680 v4 has. OpenCL is a framework for heterogeneous computing that can be used for GPU programming.

I sought to identify whether I could produce an implementation of LBM in OpenCL to run on a single GPU in BCP4 that would be faster than my MPI implementation on one node.

5.1 Original Code

Table 5: Execution times with the OpenCL implementation and speedup over the serial implementation

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

5.2 Optimisations

Table 6: Execution times with the OpenCL implementation and speedup over the prior implementation

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

6 Conclusion

References

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