Parallelising d2q9-bgk.c with MPI

George Herbert
cj19328@bristol.ac.uk

23 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 an implementation of `d2q9-bgk.c` with distributed memory parallelism that ran on multiple processes across multiple nodes—with MPI being used for interprocess communication—would provide an even more substantial performance improvement.

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.

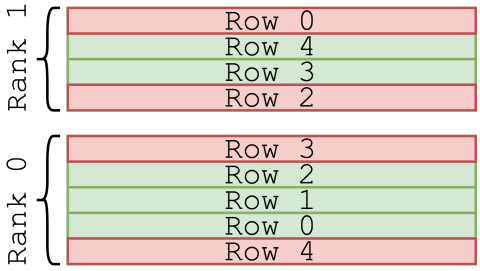

Figure 1: Row allocation example with five rows and two processes

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 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 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 serial optimised implementation, as well as my OpenMP implementation. The speedup was primarily achieved by executing sections of the `timestep` procedure in parallel, and utilising 112 times more L1 and L2 cache, and 8 times more L3 cache.

Table 1: Execution times with the 112 process MPI implementation and speedup over both my serial optimised 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 reduce 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 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.

3 Analysis

3.1 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 was assigned to a single core. I then calculated the subsequent processes' speedup over a single process implementation. Figure 2 displays the result speedup curves.

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`

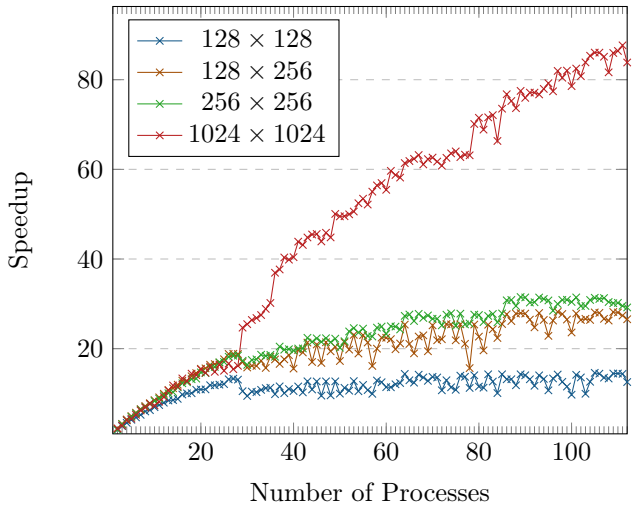

Figure 2: Speedup curves for my MPI implementation

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

My implementation followed a similar pattern for the 1024×1024 grid, however there was a clear increase in speedup after approximately 30 processes. The super-linear speedup at this point was caused by the largest problem size being split sufficiently small to fit into the L3 cache.

3.2 Comparison to Serial

4 Hybrid MPI and OpenMP

I sought to investigate whether a hybrid implementation 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 distributed memory parallelism implementation built with MPI with my shared-memory parallelism implementation built with OpenMP. Figure

3 displays the speedup curves of these two implementations with the 128×128 and the 1024×1024 grid sizes.

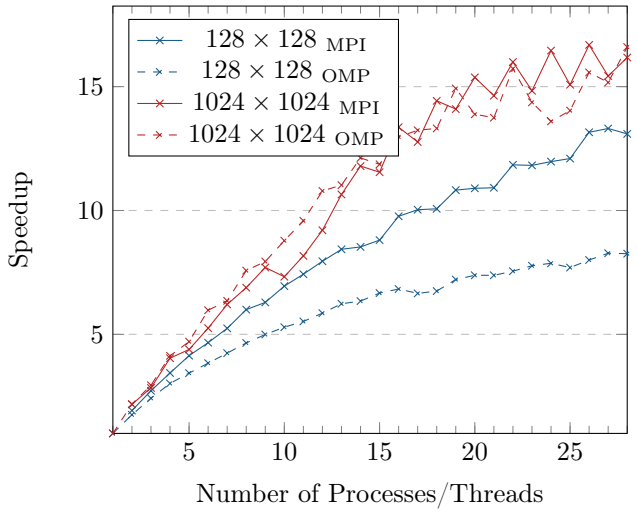

Figure 3: Speedup curves for my OpenMP and MPI implementation

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.

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

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 bandwidth, and 9.8x the peak double precision performance than two Intel E5-2840 v4 CPUs have. OpenCL is a framework for heterogeneous computing that can be used for GPU programming. I sought to produce an implementation of LBM to run on a single NVIDIA Tesla P100 to identify how fast it could run.

5.1 Implementation

I used my serial optimised 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 opted to keep my `cells` and `cells_new` data in the SoA format, since coalesced memory accesses are 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 average velocities of each cell in the same work group. I opted to implement a parallel reduction to prevent a single item in the work group

Parallel reduction.

Table 4: 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		
2048×2048		
4096×4096		

Table 5: 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		

5.2 Work-Group Size

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).