Parallelising d2q9-bgk.c with MPI

George Herbert
`cj19328@bristol.ac.uk`

17 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

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.

1.1 Hypothesis

I previously achieved a substantial performance improvement parallelising sections of my code with OpenMP. The most notable improvement came from executing 28 iterations of the inner loop in the `timestep` function in parallel. To accomplish this, OpenMP launched 28 threads, each of which ran on one of the 28 cores of a BC4 compute node. Naturally, I conjectured that an even greater performance improvement could be achieved by utilising more than 28 cores. However, OpenMP was designed for shared-memory parallelism, and so was restricted to a single node—a considerable restriction considering BC4 contained hundreds of nodes. Therefore, I hypothesised that I could use the Intel MPI Library to execute `d2q9-bgk.c` on multiple processes across multiple nodes in parallel, which would provide an even more significant performance improvement.

1.2 Implementation

I selected to use the final implementation of my program before single instruction, multiple data (SIMD) vectorization as my starting point.

Since the `cells` grid was stored in row-major order, I chose to 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 row, where y was the number of rows and n the number of processes. This was the most equal way to allocate cells to processes without splitting on a sub-row level, which I decided against to avoid increasing the complexity of my program and incurring an additional computational overhead. Additionally, since updating the value of a given cell required the values of all adjacent cells,

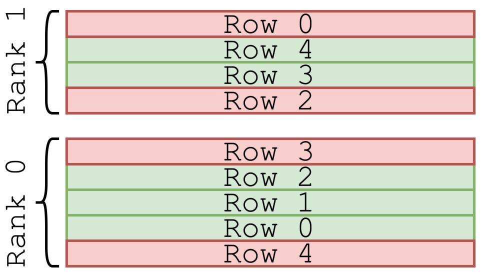

Figure 1: Row allocation example with five rows and two processes

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.

Since processes have their own memory space, I had to explicitly send the contents of cells between processes. To do so, I created a `halo_exchange` procedure. More specifically, at the end of each timestep, the bottom-most row allocated to each process was copied 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 same process was then repeated for the top-most row, which was sent to the succeeding rank.

After all iterations have been complete, each process (with the exception of rank zero) sends the final state of its `cells` grid section to rank zero with the `MPI_Send` procedure. Rank zero receives each section using the `MPI_Recv` procedure.

To compute the average velocities at each timestep.

1.3 Results

Table 1: Execution times with the 52 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			

Each time was an average of five runs on a BlueCrystal Phase 4 (BC4) compute node—a Lenovo nx360 M5, which contained two 14-core 2.4 GHz Intel E5-2680 v4 (Broadwell) CPUs and 128 GiB of RAM [1].

2 Experiments

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. However, 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 experimented with two separate approaches to send a row in the `halo_exchange` procedure. The first approach involved nine separate calls to the `MPI_Sendrecv procedure`, one for each of the nine arrays in the SoA. The second approach involved copying the cells' values in each of the nine arrays into a large buffer, followed by a single call to the `MPI_Sendrecv` procedure.

Table 2 and Table 3 display the results for the first and second approach, respectively. The second approach was significantly faster, since the overhead introduced by nine separate calls to the `MPI_Sendrecv` procedure was larger than the overhead introduced by copying the values within the nine arrays into a single buffer.

Table 2: Execution times with the first vectorization approach and speedup over the prior implementation

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

Table 3: Execution times with the second vectorization approach and speedup over the prior implementation

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

2.2 Hybrid MPI and OpenMP

I also experimented with a hybrid MPI and OpenMP implementation. In this implementation, each of the four compute nodes contained a single process.

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

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

2.3 OpenMP vs MPI

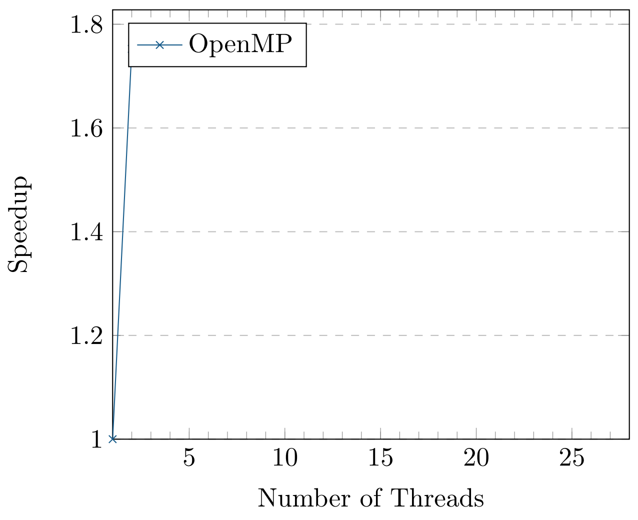

Figure 2: Speedup curves for my OpenMP and MPI implementation on the 1024×1024 grid

2.4 Scaling

3 Comparison to Serial

4 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.

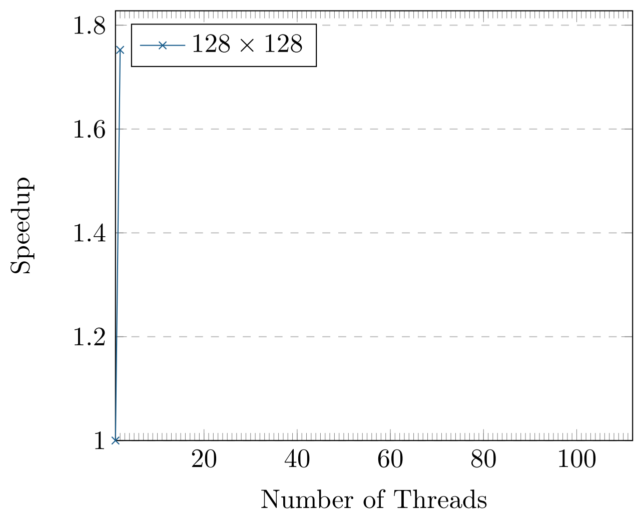

Figure 3: Speedup curves for my MPI implementation

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		

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		

4.1 Original Code

4.2 Optimisations

5 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).