Student number: 2101051

Optimizing an implementation of the D2Q9 algorithm using MPI and OpenCL

Joshua Gawley
kq21663@bristol.ac.uk

I. Introduction

In this report, we explore the parallelization of a reference implementation of the *D2Q9 algorithm* using MPI and OpenCL, following on from our optimizations in the first coursework. We should that, similar to our OpenMP implementation, we achieve worse performance than ballpark times on small problem sizes but better performance than ballpark times on larger problem sizes. In addition, we show that the OpenCL implementation is faster (sometimes significantly) than the MPI implementation. We discuss how we adapted our previous implementation to use MPI, and also some of the OpenCL-specific optimizations we made.

II. MPI Implementation

We first discuss parallelizing the implementation with MPI. Our starting point was the autovectorized implementation from the first coursework: we present the overall runtimes for that implementation below.

TABLE I: Runtimes after serial optimizations and autovectorization

Problem size	Compute time
128x128	11.5sec
128x256	23.3sec
256x256	92.9sec
1024x1024	398.2sec

A. Data initialization and domain decomposition

We first discuss our approach to initializing the data in the MPI implementation. Naïvely, we could initialize the entire grid on each MPI rank and then each rank could operate on a given subgrid. This is problematic as since each rank only operates on a subset of the grid, having each rank load the entire grid from memory is wasteful.

As such, we decided to have each rank load its own subset of the main grid, which brings us to the topic of *domain decomposition*. We could decide to distribute the grid to ranks by tiles, columns, or rows. The choice here is important because on each timestep, the algorithm requires information from not just a given cell but also the neighbouring cells in each cardinal direction. Therefore, each rank needs to access grid data from neighbouring ranks in order to process cells on the edge of a given rank's subgrid.

To do this, each rank stores a *halo region* which stores cells around the edge of a rank's particular subgrid. On every iteration, this halo region is populated with cell data from neighbouring timesteps in a process called *halo exchange*. This requires messages to be passed between processes and thus, we decide to distribute the grid to each rank by rows. This is because the autovectorized code from the first coursework uses *row-major ordering*, which means that rows are stored contiguously in memory. Hence, less messages need to be passed and less memory accesses are needed to send rows between ranks than with columns or tiles.

In terms of how many cells we distribute to each rank, for reasons of load balancing between the ranks, we choose to distribute as close to an equal number of cells to each rank. This is trivial when the heigh of the input grid is divisible by the number of MPI processes. In cases where this is not the case, we find the remainder and then distribute the remaining rows one at a time to each rank, starting from the *root* process (rank 0).

B. Halo exchange

We now move on to discuss how we implemented the halo exchange. We did so using two `MPI_Sendrecv` calls per speed array in the `t_speed` struct, where first, each process sends to its left and receives from the right, and then moves on to sending to the right and receiving from the left. The use of `MPI_Sendrecv` instead of separate `MPI_Send` and `MPI_Recv` calls results in more readable code that is also free of deadlocks.

C. Timestep calculations

We now detail the (minor) changes made to the `timestep` function. First, the `timestep` function has been changed to return the total velocity of all cells in a rank rather than the average velocity. This is because we moved the average velocity calculation after the timestep loop when we collate the velocities from each rank.

Moreover, we no longer loop over the entire grid of cells passed into `timestep`; instead, we skip the first and last rows because these are the halo regions as discussed below; on the i-th timestep, the halo regions will contain values from the $(i-1)$-th iteration but this is immaterial to the computation because the halo regions are updated at the beginning of every iteration.

D. Collating results

We now discuss how we collated the cell and velocity from the various ranks onto the root process (rank 0). We need to do so in order to calculate the Reynolds number and print this to the user at the end of the program.

We first detail how we collated the cell data from the various ranks. A naïve approach would be to do this using point-to-point communication primitives such as `MPI_Send`/`MPI_Recv` (as we did for the halo exchange). However, this would result in less performant code as we would have to send multiple messages per rank, and less readable code as we would have to write different code the root process versus other processes.

In order to sidestep these problem, we used MPI *collective* operations to collect the cells data, namely `MPI_Gatherv`, which gathers a variable quantity of data from all ranks into a single given rank (in our case, the root process, rank 0). We have to use `MPI_Gatherv` and not `MPI_Gather` as we may give a variable number of cells to each rank (as discussed earlier).

We now discuss how we collated the velocities data. In the autovectorized code from the first coursework, we calculate average velocities on each timestep as each timestep processes the entire grid. As each timestep only processes a subgrid of cells, we postpose the calculation of average velocities until the collation stage.

When we calculate average velocities, we first collate the total velocities of the cells belonging to each rank onto the root process using `MPI_Reduce`. We then go through the divide each velocity by the number of total unblocked cells to obtain the average velocity. A small optimization we make is to precalculate the number of total unblocked cells in the `initialise` function rather than during collation.

E. OpenMP directives

Finally, although not strictly necessary, we kept the `omp simd` directives from the autovectorized code in order to encourage the compiler to vectorize the code.

F. Runtimes and potential improvements

We now give the compute times for the MPI implementation (running on 4 nodes with 28 cores each).

TABLE II: Comparing compute times of MPI Implementation with ballpark times

Problem size	Ballpark compute time	Actual compute time	Speedup
128x128	0.63sec	1.00sec	0.63x
128x256	0.76sec	1.27sec	0.60x
256x256	2.70sec	3.52sec	0.77x
1024x1024	6.50sec	6.09sec	1.07x
2048x2048	15.0sec	11.1sec	1.35x

We see that similar to the OpenMP implementation, we miss the ballpark times on smaller problem sizes but beat them on larger problem sizes. In addition, we see that speedups grow larger as the problem size increases.

The next table shows the initialization and collate times for all the problem sizes (again running on 4 nodes with 28 cores each).

TABLE III: Detailed runtimes of MPI Implementation on 4 x 28 cores

Problem size	Initialization time	Compute time	Collate time	Overall runtime
128x128	0.0002sec	1.00sec	0.01sec	1.004sec
128x256	0.0003sec	1.27sec	0.004sec	1.28sec
256x256	0.001sec	3.52sec	0.01sec	3.53sec
1024x1024	0.004sec	6.09sec	0.02sec	6.12sec
2048x2048	0.01sec	11.1sec	0.06sec	11.2sec

We see that in all problem sizes the vast majority of the runtime is spent running the Lattice-Boltzmann simulation. This suggests that further optimizations such as using MPI I/O to read files is not useful for the given problem sizes, although these may be useful for larger problem sizes than what we tested on.

In terms of possible improvements, an obvious thing to try would be using a 2D Cartesian process topology rather than the 1D topology we used by default. This process topology would allow the MPI runtime to order ranks in a way that more closely represents the distribution of cells in physical space. This has the potential to improve performance in the halo exchange portion of the program, as neighbouring cells are more likely to be stored in "neighbouring" ranks (with respect to the physical topology on the hardware).

III. OpenCL implementation

We now look at our port of the D2Q9 algorithm to OpenCL so that we can run it on a GPU. We first note that the runtimes noted here are taken from a Linux lab machine and *not* BlueCrystal 4 because of aforementioned technical problems in the days before the deadline.

We first show the runtimes of the unoptimized OpenCL code.

TABLE IV: Runtimes of unoptimized OpenCL code

Problem size	Total runtime
128x128	26.1sec
128x256	50.6sec
256x256	190.3sec
1024x1024	662.1sec
2048x2048	1318.5sec

A. Porting the timestep function to a OpenCL kernel

The first change we made was to port the `timestep` function from the optimized serial code to a `computation` kernel. The original OpenCL code only has kernels for `accelerate_flow` and `propogate`, so porting the timestep code from the optimized serial version means that more of the timestep calculation is done on the GPU as opposed to the host device.

We only briefly cover the optimizations within the `computation` kernel itself as most of them were explained in detail in the first coursework. We apply loop fusion by integrating the `accelerate_flow`, `propagate`, `rebound` and `collsion` functions into the `computation` kernel. This also allows us to remove unnecessary memory copies by simply copying from either `src_cells` to `dst_cells` or vice versa. In order to do this in OpenCL, we need to make two different `computation` kernels in the source program, one which sends `src_cells` to `dst_cells` and another which sends `dst_cells` and `src_cells`, and we control which kernel is enqueued in the `timestep` function using the `src_to_dst` function parameter.

B. Memory coalescing

The second change we made was to the memory layout of the cells. The original OpenCL code (like the unoptimized serial code) stores the cell speeds in a *array of structures* layout. This prevents *memory coalescing*, where multiple memory accesses are combined into a single transaction, and which is one of the most important considerations when programming for GPUs. To achieve more memory coalescing, we change to using a *structure of arrays* layout like in the autovectorized code. However, we implement this differently; we use a simple array of floats to store the cell speeds rather than defining a separate `t_speed` structure.

```
/* Accessing a speed in OpenMP/MPI implementation */
src_cells->speed0[offset];

/* Accessing a speed in OpenCL implementation */
src_cells[(0 * grid_size) + offset];
```

Listing 1: Comparison of cell speed memory access methods

Like in the optimized serial code, using a structure of arrays layout means that the rows are stored contiguously in memory, which allows the GPU to perform memory coalescing.

C. Local reduction of average velocity calculations

The final major optimization we made was to implement local reduction in the average velocity calculation. To do this, we first create an array of floats on both the host and the global memory of the GPU to store the total velocities calculated by each work-group. We then create an array of floats in the *local* memory of each work-group on the GPU, which store the velocities of each work-item in each work-item.

The reduction then takes the form of a binary tree; half of the work-items in a work-group contribute to the first iteration of the reduction, then half of the remaining work-items in the next, and so on.

We now present two tables, one comparing the optimized OpenCL implementation with the original unoptimized OpenCL implementation, and another comparing the optimized OpenCL implementation with the MPI implementation.

We see that compared to the unoptimized OpenCL implementation, our optimizations led to anywhere between a 100x and 200x speedup, with the speedup increasing as the problem size increases. On the other hand, compared to our MPI implementation, the OpenCL implementation achieves beteen a 1.6x and 4x speedup, with the speedup decreasing as the problem size increases. This perhaps suggests that for large enough problem sizes, the MPI implementation could be more performant than the optimized OpenCL implementation.

TABLE V: Comparing optimized OpenCL implementation with unoptimized OpenCL implementation

Problem size	Unoptimized runtime	Optimized runtime	Speedup
128x128	26.1sec	0.25sec	104.4x
128x256	50.6sec	0.34sec	148.8x
256x256	190.3sec	1.10.sec	173.0x
1024x1024	662.1sec	3.73sec	177.5x
2048x2048	1318.5sec	6.93sec	190.3x

TABLE VI: Comparing optimized OpenCL implementation with MPI implementation

Problem size	MPI runtime	Optimized OpenCL runtime	Speedup
128x128	1.00sec	0.25sec	4.00x
128x256	1.27sec	0.34sec	3.74x
256x256	3.52sec	1.10.sec	3.20x
1024x1024	6.09sec	3.73sec	1.63x
2048x2048	11.1sec	6.93sec	1.60x

IV. Performance Analysis

We now analyze the performance of our MPI and OpenCL implementations. We look at various aspects such as parallel scaling and computational intensity, using a Roofline model to compare the performance characteristics of the MPI and OpenCL implementations with the various implementations from the first coursework.

A. Parallel scaling

We now analyze the scaling behaviour of our MPI implementation as we increase the number of threads. We first note that, as with the first coursework, we ended up parallelizing the vast majority of the code of the original implementation. As such, we expect a roughly linear relationship between the number of threads and the resultant speedup compared to running on a single thread.

We show the parallel scaling behaviour (Fig. 1) overleaf. We see that, as expected, the speedup increases roughly linearly as the number of threads increases. This actually contradicts what we said in the first coursework, where we expected the speedup to increase logarithmically as the number of threads increased past 32 threads. Further investigation with even larger problem sizes would be needed to determine whether the relationship between speedup and number of threads would eventually become logarithmic rather than linear.

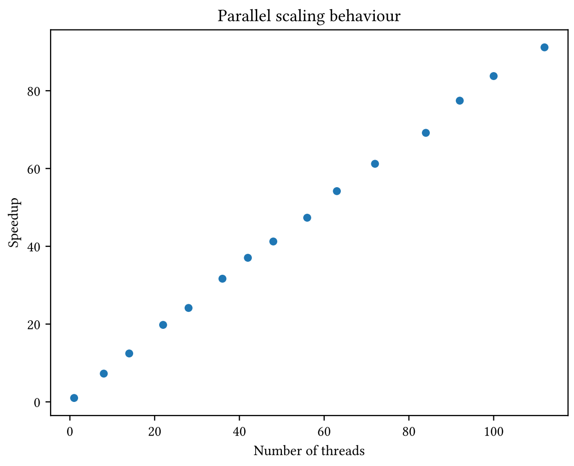

Fig. 1: Speedup vs number of threads for MPI implementation

B. Roofline analysis

Due to time constraints and lack of supercomputer availability, we were only able to collect limited Roofline data for the MPI implementation and none for the OpenCL implementation. Because the MPI implementation relies on halo exchange during the timestep loop (and so requires many main memory accesses), we would expect that the performanace of the MPI implementation is bounded by memory bandwidth.

We show the Roofline analysis for the MPI implementation (Fig. 2) and then the Roofline analysis from the pervious coursework (Fig. 3)

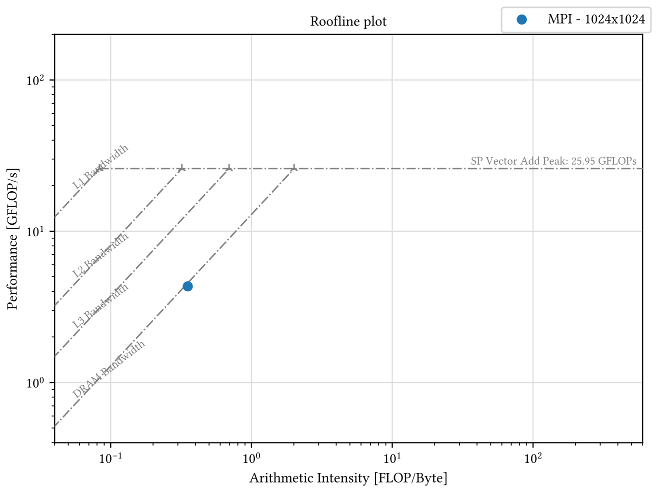

Fig. 2: Roofline analysis of rank 99 of MPI implementation running on 4 x 28

First, we note that the Roofline plots are not directly comparable as the Roofline plot for the MPI implementation shows bandwidth bounds for *single-threaded* applications whereas the Roofline plot for the first coursework shows bounds for *multi-threaded* applications.

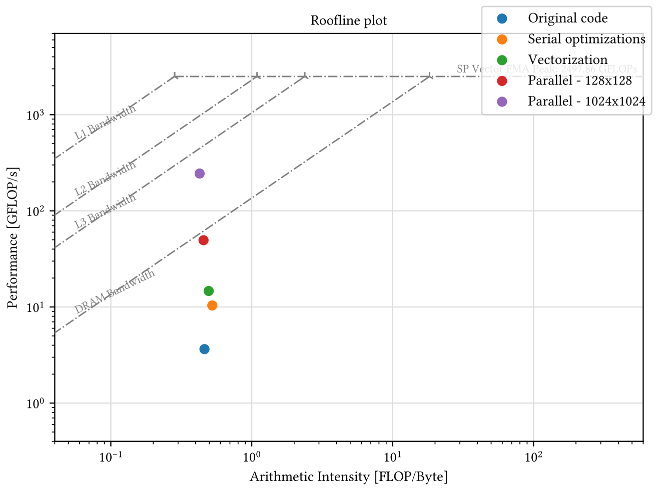

Fig. 3: Roofline analysis from first coursework

What is notable is that despite the improvement in compute time, the raw performance of the MPI implementation on a single rank is much worse than the serial implementations. This suggests that the advantage of using MPI stems from its ability to scale to a larger number of threads rather than necessarily providing the best single-threaded performance.

In addition, the arithmetic intensity of the MPI implementation is lower than the arithmetic intensity of the serial and OpenMP implementations. This is to be expected as the MPI implementation has to exchange messages on every timestep, which means that the timestep loop in the MPI implementation performs more memory accesses than the serial and OpenMP implementation.

In fact, the Roofline plot for the MPI implementation suggests that it saturates memory bandwidth and thus is bounded by memory bandwidth, as we expected. This also means that further performance improvements are likely to come from either using more performant hardware or from increasing the number of threads that the program runs on. Some tests on the author's M1 Macbook Pro suggest that using more recent hardware with increased memory bandwidth may increase performance, and it would be interesting to run this implementation on the new Isambard-AI supercomputer to confirm this hypothesis.

V. Conclusion

We ported an optimized serial implementation of the D2Q9 algorithm to use MPI and OpenCL. For the MPI implementation, we achieved slower runtimes than ballpark on smaller problem sizes and faster runtimes than ballpark on larger problem sizes, and our optimized OpenCL implementation achieved faster runtimes than the MPI implementation (and unoptimized OpenCL implementation). We analyzed the performance of both implementations by comparing runtimes, and for the MPI implementation, we additionally analyzed its parallel scaling behaviour and then used a Roofline analysis to understand the performance characteristics of the MPI implementation.