Liquid Crystals Report

Ben Morgan 25th April

# Introduction

This plot gives some insight into the vortex that was generated. The top left pane shows the density of the vortex with respect to the x and y coordinates. The bottom left shows the vorticity. The top right pane shows the velocity profile across the vortex. The bottom right plot shows how the error in density and velocity changes over time.

# Numba

I decided that first it would be interesting to try the most naive numba optimisation approach and place the numba @nb.jit handler infront of all\_energy() one energy() and get\_order(). This resulted in a nice speedup from the outset.

I ran some array jobs on blue crystal which ran the unmodified code and the numba optimised code with various grid sizes.

A graph with a line and a number of numbers

Description automatically generated

The numba code is significantly faster at all grid sizes. I calculated how many times faster the numba version was for all datapoints and found that the numba version was on average 10.06x faster than the unmodified version. I ploted these datapoints in the below graph.

A graph with blue dots and numbers

Description automatically generated

The speedup does not seem to depend too much on grid size and the datapoints and the highest and lowest speedups only vary by 5.3%.

This speedup occurs in spite of the fact that the get\_order() function reverted back to object mode (as seen in the below picture). This means that the speedup will have occurred mostly as a result of optimising the all\_energy and one\_energy functions.

A screenshot of a computer program

Description automatically generated

# MPI

I next decided to attempt to use mpi to speed up the program. I thought MC\_step would be a good candidate for mpi parallelisation as it computed values over a large grid that could be split up into smaller pieces. It was also obvious that this could be done without much communication between threads which I believed would make its implementation simple.

I wanted to test the dependency of speed on the number of cores over which the model was ran. I chose a large grid size of 500x500 and scheduled jobs over varying numbers of cores. This graph has been plotted below. A graph with blue dots

Description automatically generated

I also replotted the graph with an orange line at 909 seconds (the wall time of the unmodified model with this gridsize).

A graph showing a line of different colors

Description automatically generated with medium confidence

This shows that even on just two cores the speedup from the parallelisation outweighs the overheads.

At first as more cores are added the wall time significantly decreases but after around 8 cores were used the benefit of using more cores significantly diminished. As a result, in future MPI runs I decided to use 8 cores as I believed this gave the best balance between speedup and computational cost. Below I have plotted a graph of the wall time multiplied by the number of cores for each run. It shows that the computational expense increases in a roughly linear fasion as more cores are introduced

A graph showing a line of growth

Description automatically generated with medium confidence

At this point the MPI implementation was slower than the numba implementation, so I decided to combine the two.

# MPI + NUMBA

I parallelised my numba code with MPI and then ran this on blue crystal with successively larger grids. I used bigger jobs than before and also reran my numba code with jobs of the same size. A plot showing the speedup that was achieved is shown below.

A spA graph with blue and orange lines

Description automatically generated

An average speedup of 4.266593x was achieved once the numba code had been parallelised.

# Vectorisation

I decided to try to speed up the all\_energy function by vectorising it. Firstly I tried to use numba automatic vectorisation. I compared this with the previous numba version. The below plot shows my results.

A graph with numbers and lines

Description automatically generated

Unfortunately, the speedup was marginal at best. The numba version had a hump which did not appear in the vectorised version. It is possible that the lack of speedup occurred because my implementation is poor, or it could have happened because what I implemented was similar to what numba had already done negating any benefit.

# Vectorisation with Numpy

I tried to vectorise the code using numpy vectors. This initially worked badly and slowed my code down as seen in the plot below.

A graph with a line graph and a line graph

Description automatically generated

I used a python cProfile to produce the below table which looks at the functions I edited and attempts to interpret what produced benefits and what caused detrimental effects.

|  |  |  |
| --- | --- | --- |
|  | Unmodified Cumulative Time / s | Vectorised Cumulative Time / s |
| one\_energy | 2.086 | 3.111 |
| all\_energy | 0.694 | 0.016 |
| get\_order | 0.821 | 0.047 |

all\_energy had become 43x faster and get\_order had become 17 times faster. Unfortunately, one\_energy had become 1.5x slower and due to the program spending lots of time in one\_energy this had negated any performance benefits. I decided to revert one\_energy to the unvectorised version. The below plot shows the results.

A graph with blue and orange lines

Description automatically generated

These gave modest improvements (an average of 14% speedup) compared to the original code but were far slower than the serial numba version and slower still than the MPI parallelised numba code.

This speedup was very poor in comparison to the numba optimised code from earlier so I did not think it was worth parallelising it further with MPI

# Cython

I used cython to optimise the code by attempting to redefine all\_energy, one\_energy, and get\_order in a .pyx file and then compile them such that compiled c versions of these functions could be called by the LebwohlLasher python file.

I was able to successfully implement all\_energy and one\_energy but struggled to write a functioning version of get\_order. I decided instead to accelerate get\_order with numba but use the cython functions for all\_energy and one\_energy. This implementation ended up with some good speedup which is plotted below.

A graph with a line graph and a line graph

Description automatically generated

The average speedup was x1.45 which is good but not as good as numba so I decided that it wasn’t worth pursuing further with MPI in its current form.

# Conclusions

The fastest implementation I achieved was the version that had all\_energy, one\_energy and, get\_order optimised with numba and mc\_step parallelised with MPI. This produced a mean speedup of x23 which I was pleased with. The graph below shows the walltimes of all of the notable versions of the code across a range of grid sizes. A graph of a number of wall clocks

Description automatically generated with medium confidence