Liquid Crystals Report

Ben Morgan 7th November

# Introduction

We were given a file called LebwohlLasher.py which performed a series of calculation on a grid to model the way that a liquid crystal behaves at certain temperatures. I optimised this code in a variety of ways which I have archived in a github repository. I also validated the outputs of the optimised functions using pytest.

# 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

## ease of coding

Numba provided an incredibly easy win and sped up the code by a large amount in a way that required very little modification of the code and was very simple to implement. I did however initially run into some drawbacks as numba required specific versions of numpy to work effectively which clashed with the version I had on my computer. This meant that the numba code only worked once I had constructed a suitable virtual environment (requirements.txt can be found on my github). This took away some of the joy as otherwise it would have been an extremely easy implementation for a very nice speed up.

## Flexibility

Numba is inflexible in implementation. It worked very well on functions which were simply iterating and performing calculations but did not work on functions which were more complicated. It seemed to optimise all\_energy and one\_energy very successfully. It reverted to object mode for get\_order but still provided a speedup. Function like mc\_step was not successfully optimised with numba.

My main thought on numba was that in cases where it could easily be applied to a function it was very helpful but when a function could not be readily optimised with it, it was then quite hard to work out what needed to be done to make it work.

## Maintenance

It left code that was very readable so I think numba was good for maintenance. I did find that things which worked in one version of numba were liable not to work in other versions. This means that you would need to be very aware of updates to numba and strict with your environments.

# 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 all 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.

## ease of coding

MPI is a powerful tool which allowed the code to be effectively parallelised across many cores on blue crystal. In this case it was also easy to implement as mc\_step was highly parallelisable and could be parallelised without much communication between threads (a reduction was all that was required). Other implementations with lots of communication would be much harder to implement.

## Flexibility

MPI could be applied to most problems but may become very complicated in cases that require lots of communication and it could be easy to create locks or communicate so much that the benefits are negated. It is also worth noting that it doesn’t speed up slow code it just breaks it into small pieces and runs them over more cores. This means that if you used it to optimise code that was unnecessarily slow (e.g lots of list operations in python when you could just use numpy) it may provide a speedup but will do this at the cost of wasting loads of cpu time.

## Maintenance

MPI can make code harder to maintain as it needs to be scheduled on an appropriate machine with multiple cores. This was not possible on my laptop so it meant all runs and tests had to be performed on blue crystal which made it harder.

# MPI + NUMBA

I parallelised my numba code with MPI (8 cores) 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 when compared to the serial numba code. This constituted a brilliant speedup compared to the unmodified code.

# 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.

## ease of coding

In this case I found the numpy code quite unintuitive to write. This is not always the case though and numpy is often very intuitive to write and provides massive speedups.

## Flexibility

Numpy is pretty flexible but is definitely better for solving some problems than others. If you tried to vectorise certain functions (such as mc\_step) I think it would be quite problematic as lots of functions would need to be called. I believe this would negate a lot of the benefit

## Maintenance

The changes I made to the code made it slightly faster but, in my opinion, massively decreased its readability.

# 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.

## ease of coding

I found it confusing to use the c style function and variable definitions in the .pyx files. I found them to be a slightly weird mix of c and python. It felt like it was hard to tell where the c stopped and the python started. I am sure however that this improves with experience.

## Flexibility

I was very impressed by the flexibility of cython. It appeared that with enough effort you could turn almost any python function into a precompiled cython function and potentially win very nice speedups.

## Maintenance

Cython may be slightly harder to maintain than python code as it needs to be compiled and the compiled version of a function will only work on one machine. This will complicate things and can also mean that the speedups on one computer may be different to another depending on the architecture. As mentioned before I also found it to be a slightly confusing mix between c and python and this would make it harder for a pure python developer to maintain.

# testing

I tested the functions using a set of pytest tests which I had stored in /tests. I liked this implementation a lot as I could test all of my functions automatically with one call on the command line. I only the functions which I had edited using pairs of inputs and outputs I had generated from the unmodified code. An example of this is shown below. A computer screen with colorful text

Description automatically generated

I think this testing method worked quite well but I only covered functions I had modified. If I acidentally modified functions or there were strange unforseen consiquences these issues may have not been detected. As a result if I did this again I would expand the testing plan to emphasise a greater coverage.

# 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