Accelerating a Script Performing the Lebwohl-Lasher Process

M.T.D.R.Dolan
School of Chemistry, University of Bristol.
(Dated: March 12, 2024)

The objective for this project is to speed up a piece of code simulating the Lebwohl-Lasher process. This will be tested using Numba, Cython and vectorization methods (or combinations thereof). A parallelisation method will also be discussed. This will be successful, showing that the most effective method is a combination of vectorization and Cython which is able to speed up the code by about 3200%.

1 Introduction

1.1 The Lebwohl-Lasher model

The Lebwohl-Lasher model [1] uses a Monte-Carlo method, which relies on repeated random sampling, to simulate the behaviour of nematic liquid crystals.

Initially a random lattice is created, in which each cell is given a random orientation. The energy of cell, j, can then be calculated through interaction with each of its direct neighbours, i, given by:

$$u_j = -\frac{\epsilon}{2}\sum_i (2\cos^2(\theta_j - \theta_i) - 1) \tag{1}$$

The Metropolis algorithm [2], a type of Monte-Carlo method, can then be used to find the order in the system. For a system of size N^2, the following steps are repeated N^2 times:

1. Cell is picked at random
2. Energy, E_0, of cell is calculated
3. A random change to the angle is made
4. New energy, E_1, of cell is calculated
5. If $E_1 \leq E_0$, change is accepted immediately
6. Otherwise, change is accepted only if Boltz factor, $\exp\left(-\frac{E_1-E_0}{K_BT}\right)$ is bigger or equal than a randomly generated number between 0 and 1.

The order of the system is calculated using the largest eigenvector, S, of the liquid crystal order tensor $Q_{\alpha\beta}$, given by:

$$Q_{\alpha\beta} = \frac{1}{2}\langle 3l_{\alpha,i}l_{\beta,i} - \delta_{\alpha\beta}\rangle_i \tag{2}$$

where l is comprised of the Cartesian director of a cell.

2.1 The Initial Code

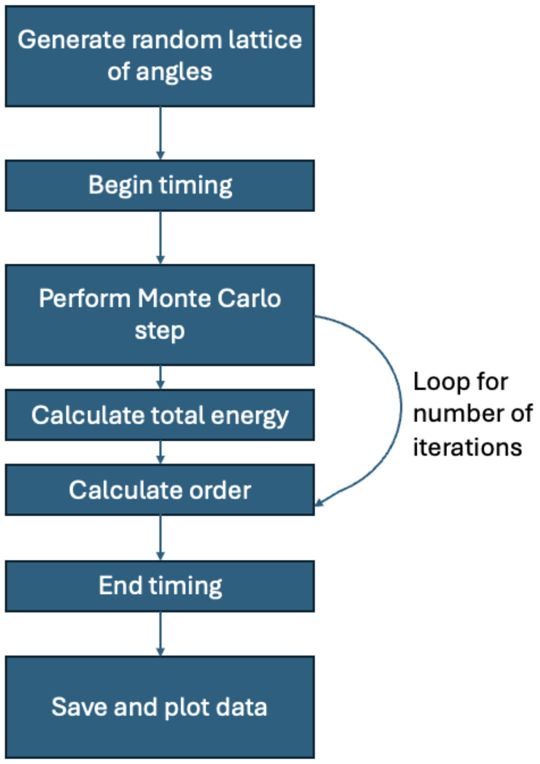

FIG. 1: A flow chart of the process originally taken by the code

Figure 1 shows the sequence in how the 'main' function of the original Lebwohl-Lasher code works. The user inputs values for the temperature, nmax (which is the length of the sides of the lattice), iterations (the number of iterations the code runs through) and pflag (which dictates if and what plot is produced).

Within the Monte Carlo and total energy functions, there is an additional function named 'one_energy' which calculates the energy of a single point in the lattice which can then be iterated over every point in the lattice.

For iterations = 50, nmax = 50 and temperature = 0.5, the code produces a rapid decrease in the Energy value from around -2500 and settles around -8000. It also produces an increase in order to around 0.38. This was used as a benchmark to test if other versions work correctly.

2 Methods

The unedited code overall took $3.232s$ to run, its major contributions are broken down in the following way:

function	total time (s)	per call (s)
one_energy	1.987	5.264×10^{-6}
get_order	0.855	0.017
MC_step	0.252	0.005
all_energy	0.026	5.098×10^{-4}

TABLE I: Breakdown of unedited code timings

2.1 Vectorization

version	execution time (s)
1	3.043
2	2.436
3	1.665
4	1.528
5	3.163

TABLE II: Times of vectorized code versions

Table 2 shows how long each version of the vectorized code took to run; each version was quicker than the previous until version 5 which was much slower than the rest.

In version 1 a new function named matrix_energy was created which was similar to one_energy but instead of having a single lattice point as an input, and giving the energy of that point as an output, it would just take the whole lattice as an input and would give an equally sized lattice with the energy of each cell as an output. In this version it iterated over the one_energy function, and so was just created to investigate where in the code an energy matrix could be used to replace a loop. This did mean that the total_energy function could be vectorized, and so still resulted in marginal acceleration of the code. This was improved in version 2 in which the matrix_energy function was itself succesfully vectorized to no longer use the one_energy function. This resulted in significantly less calls of the one_energy function, with it only now being used in the MC_step function.

Version 3 partly vectorized the get_order function to only have to run 9 times on the matrix directly instead of on each point in the matrix. Version 4 took the random number generation of the comparator for the Boltzmann factor outside of the loop in much the same way as the other random number generation of position and angle change, this resulted in a significant acceleration.

In order to cut down further on the amount of times the one_energy function was called, version 5 attempted to generate the energy matrix values outside of the loop in the MC_step and then only call the one_energy function if a point was picked twice. Ultimately, however, the addition of an if-statement in the loop made the code significantly slower overall.

2.2 Numba

version	execution time (s)
1	3.200
2	1.214
3	N/A
4	1.217

TABLE III: Times of numba code versions

version	execution time (s)
1	1.766
2	0.676
3	0.492

TABLE IV: Times of vectorized-Numba code versions

Initially in version 1, the main function was tagged with a Numba prompt, but nopython was set to false as this contained python specific methods. For the vectorized-Numba code, version 4 of the vectorized code was used as it was the fastest.

Since Numba does not work well with vectorized code, in version 2 this tag was instead changed to be above the one_energy and all_energy functions (just one_energy for the vectorized version of the code) and nopython was set to true. This resulted in a significant decrease in the time taken for the code to run. For all tests of the Numba code, it was run several times in order to allow the code to compile first and store argument types [3].

Building on this, version 4 of the just Numba code (version 3 of vectorized-Numba) set cache to be true within the tag. Interestingly this saw an improvement in speed for the vectorized code but not for the just Numba code, even if the tag was only ran on the one_energy function in both.

In Version 3 of the just Numba code, parallelisation was unsuccessfully attempted which will be expanded on in section 2.4.

2.3 Cython

version	execution time (s)
1	2.766
2	2.634
3	2.291
4	0.126

TABLE V: Times of Cython code versions

version	execution time (s)
1	0.801
2	1.052
3	0.528
4	0.517
5	0.097

TABLE VI: Times of vectorized-Cython code versions

Initially in version 1 the code was just run as is with Cython setup and run scripts, with version 4 of the vectorized script once again being used as the base for the vectorized-Cython script. This produced significant speed-ups in both scripts.

In version 2 and 3 of the Cython script, and just version 2 of the vectorized script, input variables to functions were defined. For the Cython script this consisted of floats and ints being defined in version 2, and numpy arrays being defined in version 3. In the just Cython script this caused a marginal acceleration, but it actually slowed down the vectorized-Cython script. This is likely as the increased focus on numpy vectorization in this script carried with it significant overheads.

Version 3 of the vectorized-Cython script defined all variables within functions themselves, this significantly reduced time taken in functions such as the MC_step in which so many values are calculated within the loops. For version 4, the function outputs were defined, giving very marginal speed-ups. In version 5 the cmath cos and exp functions in get_order and MC_step were used instead of their numpy counterparts. This could only be done where the function was being applied on either a float or integer, not on a whole array. This produced an incredibly large acceleration in the code, making the vectorized-Cython code the fastest out of all attempted. The combination of all these changes was applied in version 4 of the just Cython code, producing a similar level of improvement.

2.4 Parallelisation

An attempt to use mpi4py was made on the unedited script. In the main loop in which the Monte Carlo step was performed the lattice would be split into 4 different sections that could each be run through the MC_step function on a different thread, as shown below in figure 2. The results of this could then be collected by a root thread at the end of each iteration.

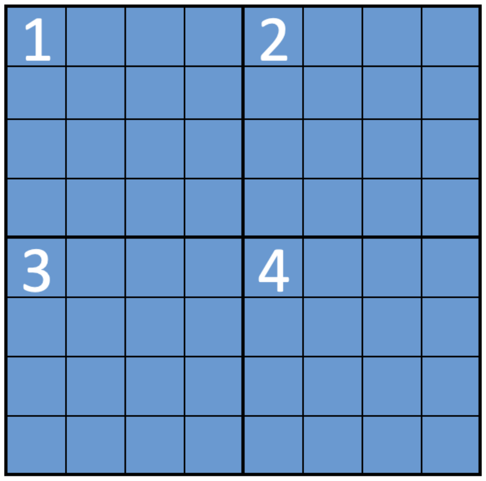

FIG. 2: A diagram of the lattice used in mpi4py Monte Carlo calculations

Unfortunately the result of this was just the whole script being run 4 times, independently on each thread. This resulted in 4 differnet calculations, taking 4 times as long.

By running the test command:

mpiexec -n 4 python -m mpi4py.bench helloworld

which returned the following message 4 times:

Hello, World! I am process 0 of 1 on eduroam-136-209.nomadic.bris.ac.uk.

it can be seen that there is a fundamental lack of communication between nodes in the mpi4py environment, and so it is unable to be effective. This error was not able to be solved.

Similarly, OpenMP parallelisation testing was done within the Numba and Cython code. However, the default MacOS compiler does not support OpenMP [4].

3 Results

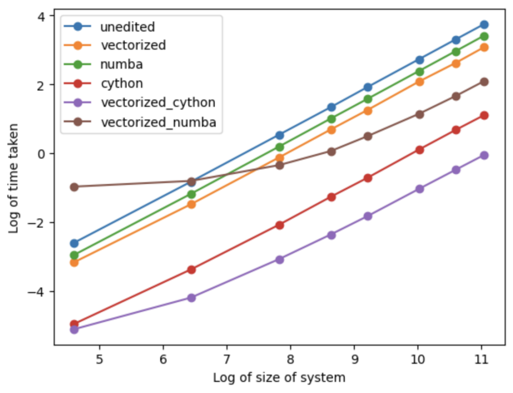

FIG. 3: Log graph showing the system size versus the time taken, iterations = 50

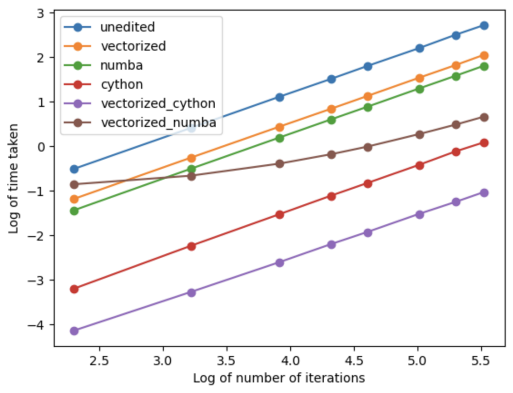

FIG. 4: Log graph showing the iterations versus the time taken, nmax = 50

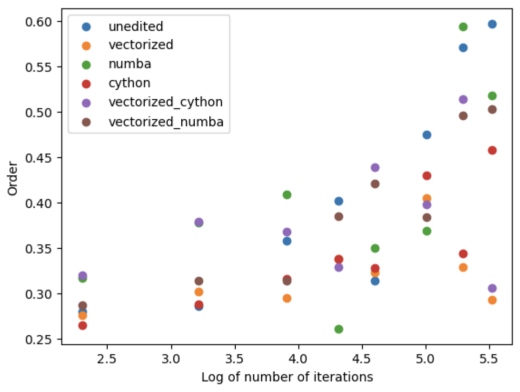

FIG. 5: Scatter plot of order versus the log of the iterations of the system

4 Conclusion

Figure 3 and 4 both show that Cython is the fastest out of the methods, with Cython vectorization being the fastest overall. For a very small system (nmax = 10) they have almost identical execution times. This is likely because the vectorization will not be making much improvement over a looping method, and for both, the overheads of Cython will be the dominant force in time of execution. Similarly for both a small system size (nmax = 10, 25), and low number of iterations (iterations = 10, 25), the execution time of vectorized Numba does not increase with increases in size or iterations. Again, this is because at these points, the dominant force in the time of execution is the Numba overhead. Interestingly this initially makes it slower than the just Numba and vectorized scripts, and even the unedited script in the system size tests, before becoming quicker once the system becomes complex before. The just Numba script does not display this behaviour because it does not have as many vectorizations to deal with.

Overall, in terms of difficulty of implementation, Numba certainly came out on top since it only requires the addition of one or two lines of code to tag relevant functions for just-in-time (JIT) compilation. Cython is also relatively easy to implement, requiring only the addition of a setup and run script in its purest form. With straightforward additions (variable definitions, cmath functions etc.) it achieved a speed-up of almost 3200%. Vectorization is the only one of the methods that requires some actual ingenuity, the restructuring of the code can be difficult in places - that being said, it should be done anyway as a matter of good 'practice'.

4 References

[1] Lebwohl, P.A. and Lasher, G., "Nematic-Liquid-Crystal Order—A Monte Carlo Calculation", Phys. Rev. A, 6, 426–429 (1972)

[2] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller and E. Teller, J. Chem. Phys. 21, 1087 (1953)

[3] Numba Documentation, https://numba.pydata.org/numba-doc/latest/user/5minguide.html, (accessed March 2024)

[4] Chrys Woods, https://chryswoods.com/accelerating_python/ parallel_cython.html, (accessed March 2024)