Analysing Methods for Accelerating Python Code

Matthew Parker

Abstract

This report analyses the effectiveness of several different methods of accelerating a provided Python script. The script produces a simulation of a 2D liquid crystal lattice over the course of multiple Monte Carlo steps, using the Lebwohl-Lasher method to calculate the interaction energies between cells in the lattice. Cython was determined as the optimal method to accelerate code, particularly when parallelised. Numba was also found to be effective. However, MPI was determined as a method which does not provide significant enough accelerations for the time put in to implement it effectively.

Introduction

Liquid crystals are materials that have properties between that of standard liquids and crystal solids. They are able to flow like a liquid but also have significant structural/molecular order. The molecules which form liquid crystals often share similar properties, with an elongated shape, a rigid backbone of double bonds, and strong dipoles.[1] These structural properties can clearly be seen in figure 1 which shows 5CPUF, an example of a molecule which forms a liquid crystal.

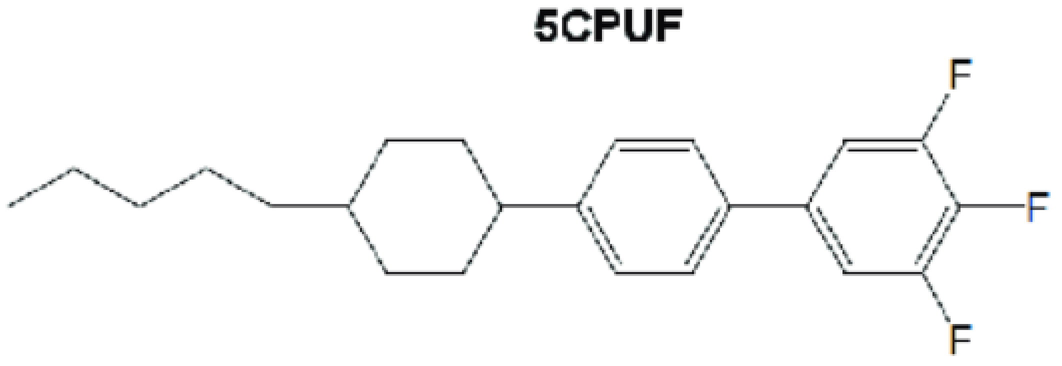

Figure 1: 5CPUF, an example of a molecule which forms liquid crystals.[2]

There are two types of liquid crystals, thermotropic, which change phase depending on temperature, and lyotropic, which form in the presence of a solvent such as water.[3] This report focusses on thermotropic liquid crystals and how they are affected by temperature.

Thermotropic liquid crystals transition through different phases depending on the temperature, with the molecules having different orientations and positions between phases. Crystal liquids progress from the isotropic phase at higher temperatures, where they have more liquid-like properties, to the crystalline phase at lower temperatures, where the molecules are highly ordered. The nematic phase represents the transition between these two phases, with a balance between the liquid and solid properties. In this phase, position is ordered at low ranges, with multiple patches of highly ordered molecules within the liquid crystal. The molecules often align in the same direction as their neighbours but retain some of their fluidity.

The fact that liquid crystals have different levels of order depending on the temperature is key to their commercial uses. Liquid crystal displays (LCDs) are commonly used in digital clocks. They have many advantages, particularly in battery-powered items, as they have very low power consumption because the molecules align very easily under low current.[4] However, they can have low contrast and colour saturation compared to other display technologies.[5] Furthermore, at high temperatures these displays can break down, as the liquid crystals become more isotropic and lose their molecular order.

Liquid crystals also occur naturally in nature, for example in proteins and cell membranes. For example, DNA can enter the liquid crystal phase. This is advantageous as it allows efficient packing of genetic material within a cell.[6]

This report involves a program to simulate a 2D liquid crystal model in the nematic phase. The Lebwohl-Lasher model is used to simulate the development of the orientation of the liquid crystal. The energy of a cell is calculated in the Lebwohl-Lasher by determining the interaction energy term, u_j, calculated using equation 1:

$$u_j = -\frac{\epsilon}{2}\sum_{i=1}^{n}\left(2\cos^2(\theta_j - \theta_i) - 1\right) \tag{1}$$

where the sum is over all adjacent cells, ε is the maximum interaction energy between pairs of adjacent liquid crystal cells, θ_j is the orientation of the cell of interest, and θ_i is the orientation of the adjacent cells.

The Metropolis Monte Carlo algorithm was used to determine each step of the simulation. This involves picking a cell at random and calculating its energy, E_0, using equation 1. Then the angle of this cell is changed by a random amount and its new energy, E_1, calculated. The random angle change is accepted and kept if E_1 is lower than E_2. If E_1 is greater than E_0, the Boltzmann factor for the energy difference is calculated, using equation 2:

$$\exp\left(-\frac{E_1 - E_0}{k_B T}\right) \tag{2}$$

where T is the temperature of the system.

This value is compared to a randomly generated number between 0 and 1, and if the Boltzmann factor is greater than the random number, the random angle change is accepted. If neither of these conditions are satisfied, the change to the angle is undone, and the cell returned to its original orientation.

One Monte Carlo step (MCS) is completed when this process is repeated the same number of times as the number of cells in the lattice.

In the methods described below, this process is sometimes changed slightly to accelerate the speed of the simulation. The selection of the cells is not done randomly, as in the original script, with two other methods used. One method involves iterating through whole rows at a time, changing the angle of the first cell in the row, accepting/rejecting the change, and then proceeding to the next cell in the row along the entire row. This means every cell is selected exactly once in a single MCS, which is not random but allows parallelisation of the calculation, as different rows can be calculated simultaneously. However, it is important to avoid calculating adjacent rows at the same time, as the energy of a cells depends on its adjacent cells.

The other way of accelerating the calculation is through the checkerboard method, which again allows efficient parallelisation. This involves calculating the energies of all values corresponding to white squares on a checkerboard first, before calculating the energies of all the black cells on a checkerboard, using the updated energy values of the white squares. This allows the whole board to be calculated in just two steps, without running the risk of adjacent cells being calculated at once.

Methods

Figure 2 shows three plots of a simulated section of a liquid crystal, with the cells coloured by their angle of orientation. Cells aligned in the same direction therefore have the same colour. It can clearly be seen that as the temperature increases, the order of the lattice decreases, with neighbouring molecules less likely to be aligned.

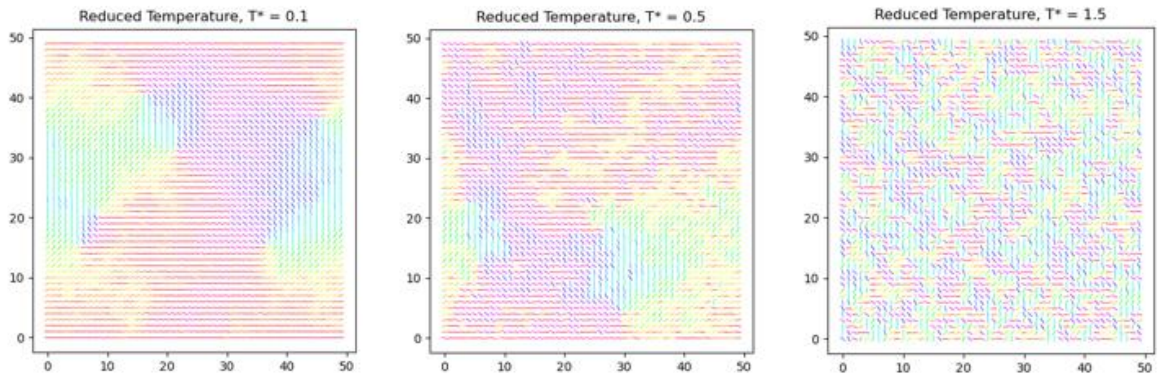

Figure 2: Figure showing the orientation of the liquid crystal cells after 1000 Monte Carlo steps at reduced temperatures of 0.1, 0.5, and 1.5. Each cell is coloured by orientation to easily show the alignment of the cells at each temperature.

Figure 3 shows how the energy varies as the simulation progresses. The energy rapidly becomes more negative up until it reaches an equilibrium, at which point the energy oscillates around this equilibrium energy for the rest of the simulation.

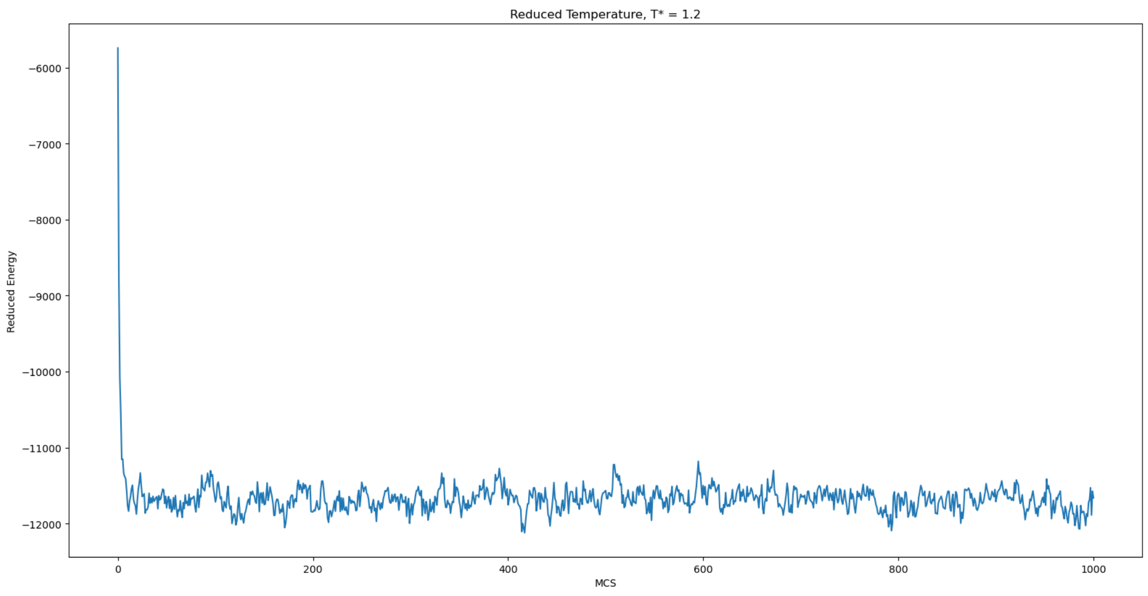

Figure 3: Energy of system over 1000 Monte Carlo steps in a 75x75 size lattice with reduced temperature set to 1.2. Once equilibrium is reached, the energy values oscillate around the equilibrium energy.

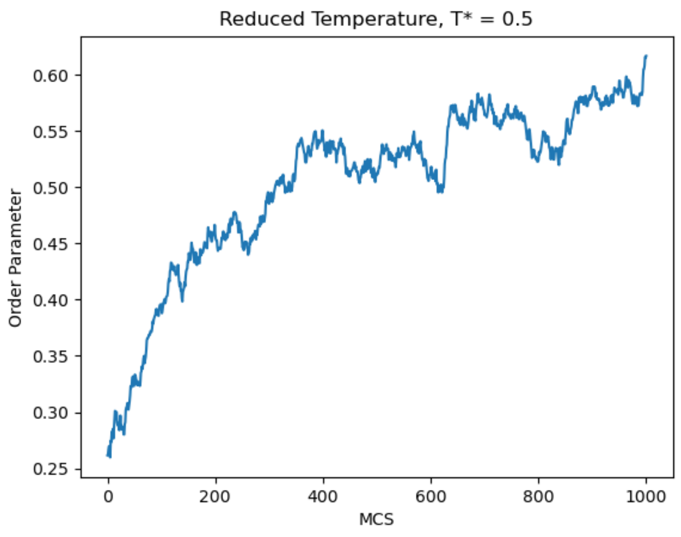

Figure 4: Example plot of how order varies over the course of a 1000-step Monte Carlo simulation in a 50x50 lattice. Reduced temperature set to 0.5.

Figure 4 shows how the order varies throughout an example simulation. As time progresses, the order parameter increases, which shows how the liquid crystal is becoming more ordered.

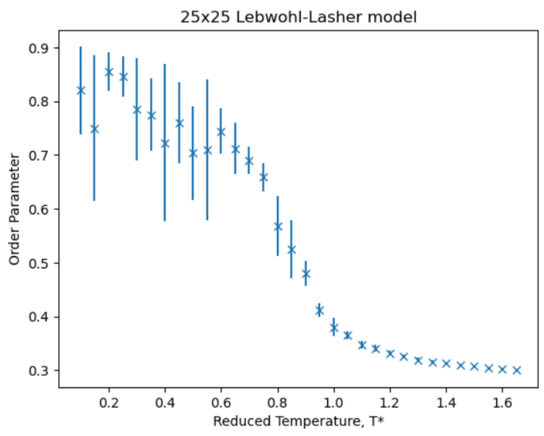

Figure 5: Order against reduced temperature for a 25x25 lattice, with the order parameter calculated after 1000 steps, and calculated as the average of 10 repeats.

Figure 5 shows how the order parameter varies with increasing temperature for an example 25x25 size lattice. At higher temperatures, the order parameter is significantly lower. This shows that the molecules within the simulated liquid crystal are significantly less ordered at higher temperatures. This is expected as liquid crystals have a more liquid-like, unstructured structure at higher temperatures, when they are in the isotropic phase.

Acceleration methods

The original script for running these simulations was very slow, with multiple nested loops iterating through the whole lattice. When the size of the lattice was set to be relatively large, the script would take an extremely long time to run.

Multiple different methods were used to accelerate the scripts, with some methods also being combined together to optimise efficiency.

The most basic method used was to vectorise the code using NumPy vectorisation. This involved removing the iterations of the simulation through each cell of the lattice, instead calculating every cell of an array at once. The checkerboard method was used in calculating the MCS for the lattice. This allowed the calculation to be done in 2 steps rather than iterating through all 2500 cells (in a 50x50 lattice).

This method was accelerated further using Numba, which allowed compilation of many of the functions, so they did not have to be recompiled every time the functions were called. Furthermore, this was parallelised which allowed the elements in the arrays to be calculated simultaneously.

Cython was another key method used to accelerate the script. This allowed compilation of multiple functions into C/C++ code. In these scripts, the original nested loops were kept, rather than using the NumPy vectorisation method. This is because Cython does not handle arrays well, and can run nested loops much faster than Python, due to the variable types being declared. To achieve the optimal speed-ups, the datatypes of the variables in the Cythonised functions must be declared. Furthermore, significant acceleration was observed when the in-built C maths functions were used rather than the NumPy versions.

A separate script was also produced to parallelise the Cython scripts, to allow multiple rows to be calculated simultaneously during each Monte Carlo step. In the parallelised Cython version, odd numbered rows were calculated first, then the even numbered rows. This ensures that adjacent rows were not being calculated simultaneously.

The final method used was using Message Passing Interface (MPI). This allows multiple processors to work on different parts of the same array, before combining the final results of each worker at the end of the required number of steps.

MPI was also used in tandem with Cython to achieve further speed-ups.

Simulation results

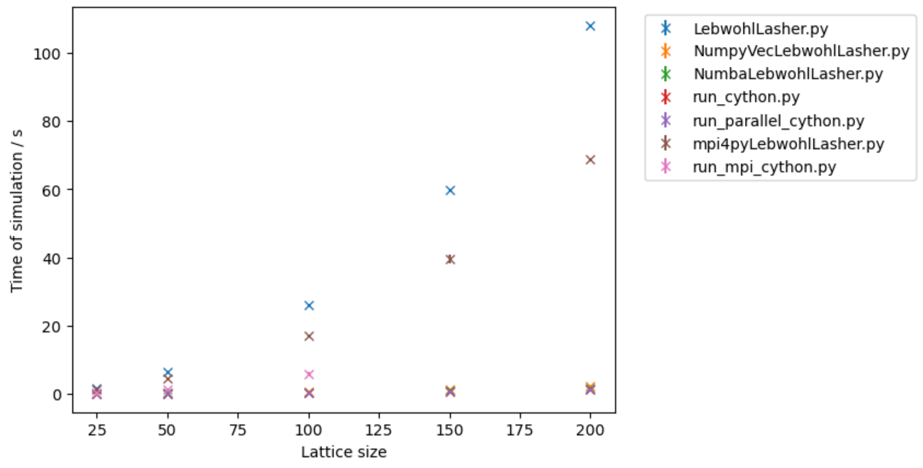

Figure 6: Average run time for each script to run 100 MC steps at varying lattice sizes.

Figure 6 shows how the time taken to run each script varies as lattice size increases. It can clearly be seen that the original script is very slow, especially as lattice size increases. The MPI script (using 4 threads) offers some but very little acceleration. However, the other scripts achieve significant speed-ups, particularly as lattice size increases.

The MPI script was based very closely on the original script, as it was hypothesised that that the MPI script would require plenty of work to improve. This is because the more work each worker has, the smaller the effect of the overhead time of sending between workers. A major limitation of using MPI to accelerate the code is that the maximum speedup is limited by the number of available threads. For example, if there are 4 threads available, the MPI method only offers a maximum acceleration of x4. This is significantly less than the acceleration offered by the other methods. Furthermore, MPI is time-consuming and difficult to implement, as it requires plenty of thought about when to effectively and accurately send and receive information between workers. Overall, the benefits of using MPI are largely outweighed by the negatives, in particular its low ceiling for accelerations, and difficulty to implement.

The efficiency of the script using MPI can be improved by combining it with other methods, such as Cython. The improvement in performance can be observed in figure 7, which shows time against lattice size for the MPI script based on the original script, and one that had part of the MC step calculation Cythonised.

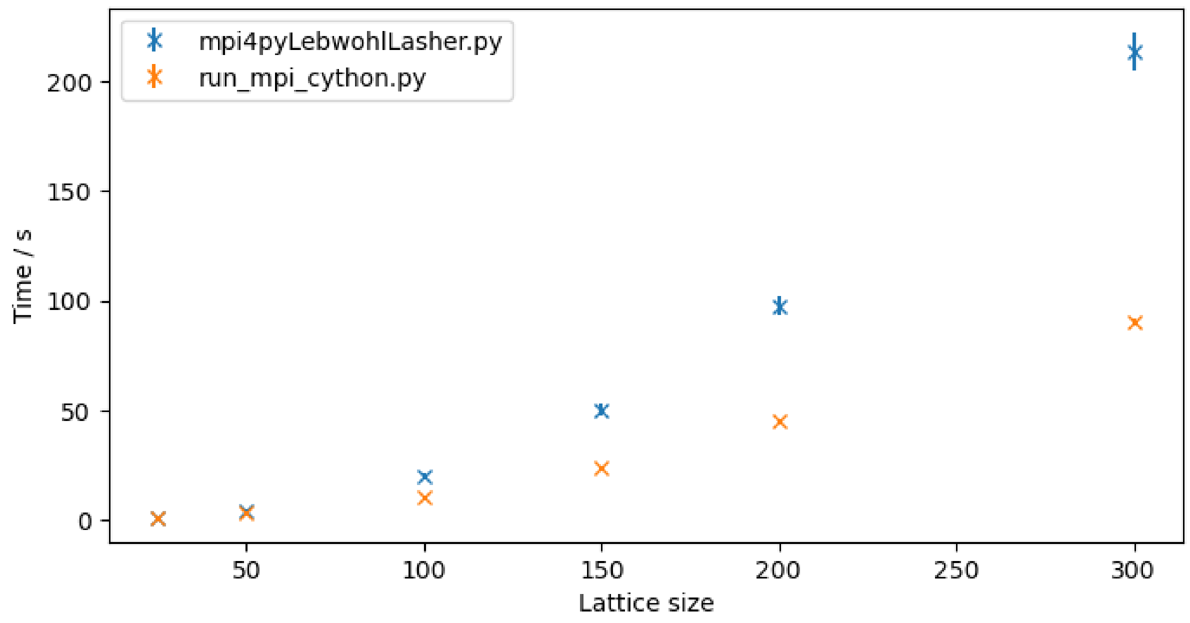

Figure 7: Comparison of script runtimes against different lattice sizes, for the MPI script based on the original code and the MPI script with one function Cythonised.

There is evidently significant improvement from using Cython combined with MPI, particularly as this was just one function that was Cythonised. However, the improvement is unrelated to MPI. MPI is a time-consuming method that requires careful thought about when to send and receive data between workers. Overall, the improvement when using MPI is not significant enough to justify using it. The other options for accelerating the provided code are much simpler methods that improve the speed of the script with significantly more and with more ease than MPI.

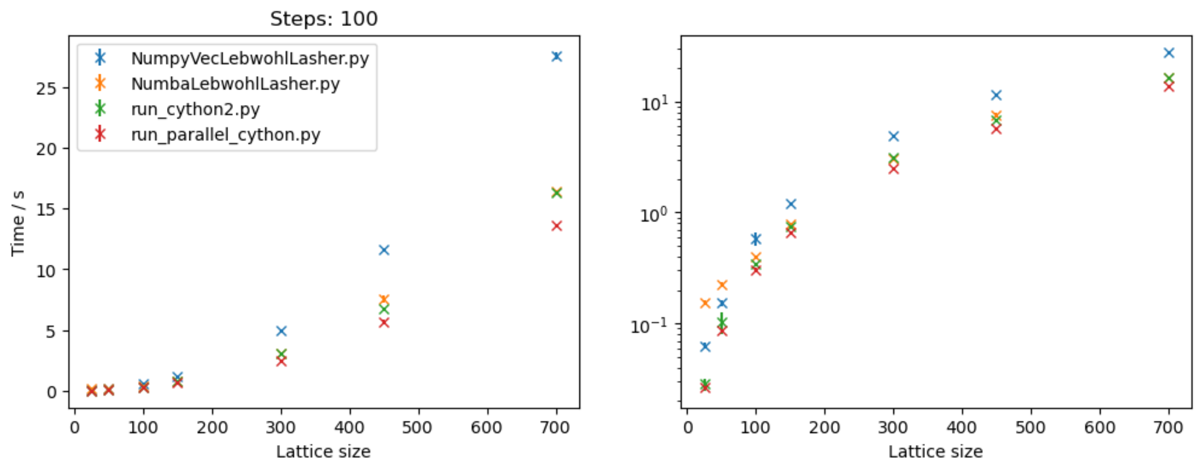

Figure 8: Comparison of run times of the fastest scripts over lattice size, with time shown both linearly and logarithmically.

Figure 8 shows how the speeds of the fastest set of scripts vary with increasing sizes of lattice. The time for each script to run an increasing number of steps increases linearly, as each step has to be calculated in order and cannot be parallelised. The variation of time with lattice size is more interesting, as different methods perform better than other methods depending on the lattice size. For example, the NumPy vectorised script outperforms the Numba script with small lattices but performs worse as lattice size increases. This is likely due to Numba being parallelised, as parallelisation is more effective when each processor has more work to do, as the effect of the overhead is reduced.

The Cython scripts generally outperform the NumPy and Numba scripts. For larger lattices, the parallel Cython script outperforms the single-threaded Cython script. However, at smaller lattices, the single-threaded Cython script outperforms the parallelised version. This is again likely because the parallel script is limited by the overhead of sending between threads.

Both the parallel Cython script and the Numba script can be parallelised. The number of threads to run the parallelised scripts can be varied. Figure 9 shows the effect of changing the number of threads on the speed of running the Numba script for different sizes of lattices.

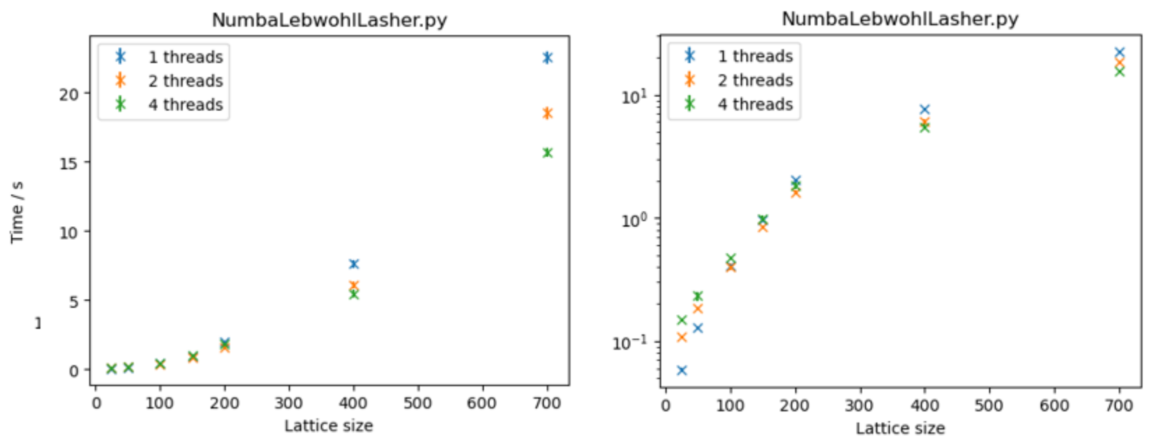

Figure 9: Comparison of the runtimes of the Numba scripts with differing number of threads. The right hand plot shows the time on a logarithmic scale to show the differences in the runtimes between the threads for smaller lattices more clearly.

It can be observed that when the lattice size is small, Numba runs most efficiently with fewer threads. However, as lattice size increases, an increased number of threads makes the simulations more efficient. This shows that using more threads is most effective when each processor has more work to do, as it reduces the effect of the overhead of parallelisation.

Conclusion

In conclusion, Cython is the most effective method for accelerating code, particularly when parallelised. Numba was also effective, particularly because it could be parallelised effectively.

Overall, Cython and Numba are both effective at accelerating code and can both be easily parallelised and implemented. The method implemented should depend on the code being accelerated, as it would be easier to optimise vectorised code using Numba, whereas Cython is easier to use on scripts which use loops. If either option is plausible, Cython would be the recommended option.

MPI is another option but is not recommended as it is time-consuming and difficult to implement and does not produce particularly impressive speed-ups by itself. It can be combined with other methods such as Cython and Numba, but the accelerations from the combination of this methods would not be from MPI. However, the benefit of it is that you get to control exactly what is done by each worker, so it can be manually optimised to use each worker to its maximum potential.

The best option for accelerating code is likely to rewrite it in C++. This was not done in this report but would produce significant acceleration compared to the methods used in this script. However, rewriting a whole script is time-consuming and not always possible. Therefore, Cython is generally the most effective and implementable method for accelerating a provided Python script.

Appendix

The Github repository for this project can be found at:
https://github.com/M4ffff/accelerating_code_assess/

References

[1] M. Stephen, J. Straley, *Rev. Mod. Phys.*, 1974, **46**, 617.
[2] Structure Of Liquid Crystal You Should Know, Daken Chem, https://www.dakenchem.com/structure-of-liquid-crystal-you-should-know/ (accessed 22/02/2025).
[3] G. Vertogen, W. De Jeu, *Thermotropic liquid crystals, fundamentals*, 2012, **45**, Springer Science & Business Media.
[4] M. Schadt, *Annual Review of Materials Research*, 1997, **27**, 305-379.
[5] H. Chen, J-H. Lee, B. Lin, *Light: Science and Applications*, 2018, **7**, 17168.
[6] T Strzelecka, M Davidson, R Rill, *Nature*, 1988, **331**, 457–460.