Accelerating the Lebwohl Lasher Python simulation

February 26, 2025

Abstract

This reports explores various methods to accelerate the Pytohn implementation of the Lebwohl Lasher model. Profiling the original code showed that the main performance bottleneck was arising from repeated energy calculations and Monte Carlo steps. The methods used to accelerate the code include NumPy vectorisation, Numba, Cython, and MPI. While Numpy vectorisation alone can reduce the runtime by an order of magnitude, Numba and CYthon - especially parallelised - achieve even larger speedups. On a 50x50 lattice, these can reduce the total time to about 0.03s compared to the original time of 4s. For larger lattices, MPI becomes increasingly beneficial, however at a cost of implementation compelxity. Overall, Numba emerged as the most straighforward and powerful option for small to medium problem sizes.

The code for this project can be found at https://github.com/nicolaegues/LL$_{a}$*cceleration*

1 Introduction

The Lebwohl-Lasher model is a Monte Carlo simulation designed to study the ordering behaviour of liquid crystals. The python implementation of this model used in this report simulates a two-dimensional square lattice of molecules, which are each represented by an orientation angle. During every Monte Carlo step, each molecule experiences random orientation changes, whereby the acceptance criteria is based on minimisatin of the energy and Boltzman probability. During the simulation, the total energy of the lattice and the order parameter are recorded over time.

However, the simulation can be very computationally expensive - especially as the problem size and iterations are increased. This report will therefore focus on exploring different methods of accelerating the python code.

Profiling the original Lebwohl-Lasher simulation code (for 50 iterations, a lattice size of 50x50, and temperature of 0.5) shows that the main performance bottlenecks are the functions that calculate the individual cell energies and perform the Monte Carlo (MC) steps. Although a single call to the cell energy function is relatively fast (about 7.57 μs), it is called around 377,500 times per simulation, leading to a significant total runtime of about 2.9 seconds. In contrast, the MC step function is only called 50 times, but each step is very computationally heavy, taking about 51.5 ms per call and 2.6 seconds in total. This is due to the numerous times the energy function is called, as each step involves calculating the energy changes for randomly selected sites.

Additionally, the function to calculate the order parameter of the lattice is also quite expensive, is called 51 times and each call takes, on average, about 17 ms - accumulating around 0.85s of runtime.

Therefore, modifying the code such that the calls to the cell-energy function are reduced, as well as optimising both this function and the MC step function should significantly accelerate the simulation.

2 Methods

To ensure that all of the below methods continued producing correct simulation results, the primary testing strategy involved producing plots of the collected energy and order values. While the energy curve should reach a low plateau, the order graph should plateau at a value close to 1. Additionally, the random number generators were seeded to allow for reproducibility, and the original raw energy output values saved. These could then be compared amongst methods - however, this testing strategy was not applicable to various appraoches - most relevantly the parallel ones.

2.1 NumPy Vectorisation

One of the most effective ways of accelerating Python code is to replace explicit for loops with Numpy vectorized operations. By using optimised C and Fortran libraries under the hood, and efficient memory access patterns, vectorized numpy can apply mathematical functions to entire arrays at once, which reduces the overhead that arises from looping.

To calculate the total energy of the lattice, for instance, the original code iterates over each lattice site, calling the cell energy function each time - one can thus expect vectorisation to speed this up a lot.

Most importantly, the MC step function is restructured to also allow for this vectorisation. Since the energy of a given cell in the lattice is determined relative to the orientations of its neighbours, any change in a neighbour's orientation will affect the local energy landscape. This means the order of updates matters - and we can't just vectorise all angle changes simultaneously. Otherwise only the "old" states would be used, leading to different simulation dynamics. To include the vectorisation, a checkerboard approach is therefore implemented. This entails first updating one set of diagonals (the "white" set) in a vectorized way, thus excluding each cell's neighbours (the complementary set). The "black" set is then updated after.

Finally, the function to calculate the order parameter of the lattice, which originally includes various nested loops, is also vectorised. This is done using numpy's einsum (einstein summation) function, which can efficiently compute the Q tensor by summing over all positions in the lattice.

2.2 Numba and Parallel Numba

Numba is a just-in-time compiler of python code - meaning it compiles the functions in the script that have certain decorators - such as @jit - into machine code at runtime. This machine code is then cached by numba for subsequent runs, which are then executed at machine-code speed - the first set of timings are therefore ignored.

The best performances are usually achieved when the "nopython" mode is set, which ensures that the compiled function does not rely on the Python interpreter (reducing overhead).

Additionally, one can easily parallelize loops by adding the "parallel = True" argument to the decorator as well as using "prange" instead of range. Numba then automatically splits the loop across CPU cores.

2.3 Cython and Parallel Cython

Cython serves as a superset to Python through which a python-like script can be automatically translated into fast C code. A setup script is used to convert the (mostly) pythonic code into a compiled binary library, which can then be imported and used within Python.

In the cython script, "cdef" can be used to declare variables and numpy array types - thus eliminating python's dynamic typing overhead. Additionally, certain C functions can directly be imported.

The MC step function, the cell energy function, the total energy function and the order parameter calculation were cythonized in this way.

Further, Cython provides a prange (parallel range) function through OpenMP, which distributes iterations actoss CPU cores. The "nogil" argument ensures that Python's Global Interpreter Lock (GIL) is released, allowing parallel execution of threads without bottlenecks. To fully make use of this, the MC step and cell energy function had to be modified to be completely GIL-independent.

2.4 MPI

MPI (Message Passing Interface) follows the SPMD model - single program, multiple data. Here, it's used to divide the simulation work among one master process and several worker processes. The master thread initialised the overall lattice and splits it into row-blocks, sending one to each worker. Each worker then performs the Monte Carlo updates locally on its section. commmunicating with neighbouring workers to exchange boundary rows (with boundary conditions being met). ! Through non-blocking communications (MPI.Isend and Irecv) these updates can happen in parallel. Then, after completing the steps, the local energies, order parameters and acceptance ratios are gathered from the workers - with the energies being summed and the average taken for the latter two.

3 Results and Discussion

System	Original	NumPy vec	Numba	Numba 4 cores	Cython	Cython 4 cores	MPI+numpy 4 cores
i5-1135G7	3.923558	0.095239	0.078701	0.044730	0.161587	0.031338	0.099195
BC4	4.368394	0.139137	0.079501	0.029122	0.092153	0.037007	0.092153

Table 1: Runtimes (in seconds) after 50 iterations, with a lattice size of 50x50 and reduced temperature T* 0.5

Table 1 shows how each acceleration method managed to significantly speed up the original python code. For a simulation involving 50 Monte Carlo steps, a lattice size of 50x50 and a reduced temperature of 0.5, the fastest methods are hereby parallel Numba (on BlueCrystal) and parallel Cython (on an i5-1135G7 processor), with runtimes of 0.029s and 0.031s, respectively. Amongst the serial methods, Numba is the fastest, with a runtime of 0.079s as run on the laptop.

The NumPy vectorisation method also produced very fast results, with a runtime of 0.085s on the i5-1135G7.

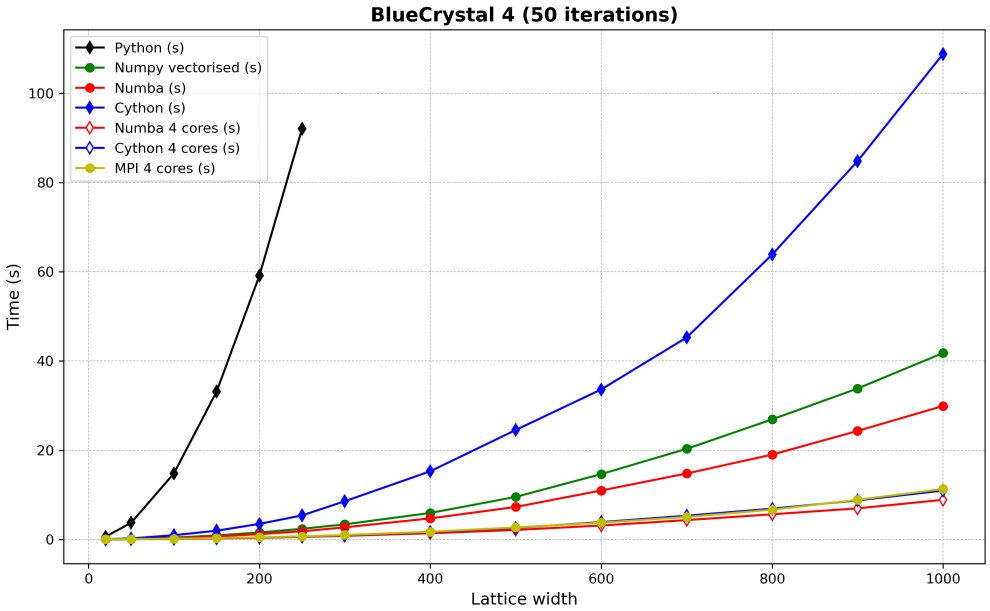

Figure 1: Runtimes in seconds for all tested methods as a function of increasing lattice size (here shown as the width of the quadratic lattice. The simulations were run for 50 Monte Carlo steps, and reduced temperature of 0.65, on Blue Crystal 4.

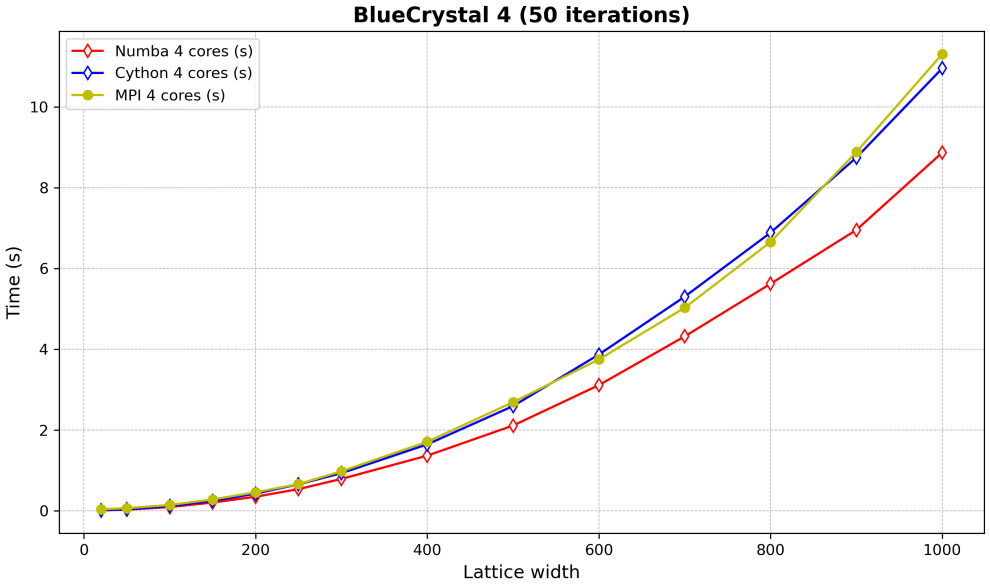

Figure 2: Runtimes in seconds for the parallel methods as a function of increasing lattice size (here shown as the width of the quadratic lattice. The simulations were run for 50 Monte Carlo steps, and reduced temperature of 0.65, on Blue Crystal 4.

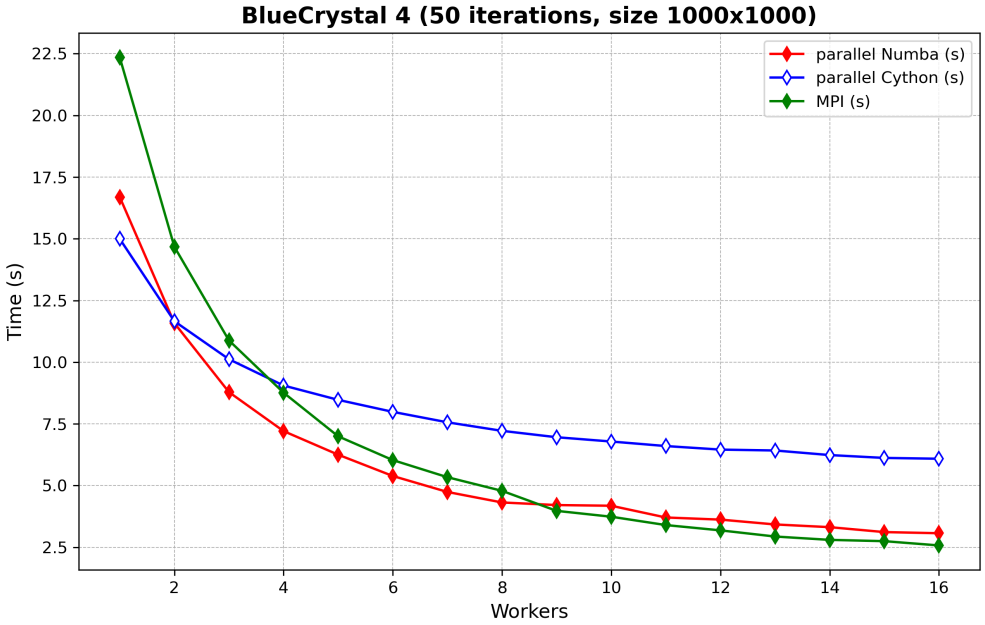

Figure 3: Runtimes in seconds for the parallel methods as a function of icreasing number of workers. Each simulation was run over 50 iterations, with a lattice size of 1000x1000 and reduced temperature of 0.65.

Replacing the total energy caculations with a vectorised energy calculation operation initially halved the runtime (to about 2.17s). The order parameter vectorisation further reduced the runtime to about 1.729 seconds. However, the biggest change came from adjusting the MC step function to allow for vectorisation via the checkerboard approach, leading to the final recorded runtime.

The further reductions in runtime by serial and parallel Numba indicated that this method is particularly well suited for the simulation, efficiently optimising the loop-heavy parts of the code. The ease with which this method can be implemented - merely by including certain decorators - makes this method particularly appealing. NumPy, in contrast, required various code modifications - including the restructuring of the MC step function - which makes it slighly less appealing. However, once implemented, it scales very well to larger lattice sizes due to the vectorized operations, as well as enhacing the overall readability of the code.

Both the serial and parallel Cython methods demonstrate that significant performance gains can be achieved by translating critical functions into compiled C code. However, there is a significant trade-off with implementation complexity. Cython requires a very careful restructuring of the code, which includes making certain sections safe for parallel execution. This, again, stands in contrast with Numba, which does not require running an additional compilation step and large code modifications.

As a distributed-memory approach, MPI is ideal for very large scale problems as it can distribute the workliad across multiple workers. The need to communicate boundary data, however, introduces a communication overhead that may not be justified for smaller lattices, as evidenced by the runtimes in table 1, where MPI+NumPy do not outperform the other methods.

Figure 1 shows how all methods scale roughly quadratically with the lattice size. The slopes differ significantly, however, due to some methods being able to reduce overheads or distribute the work amongst workers. Due to the original code's large overhead of the nested for-loops, its curve is bar far the steepest.

While Numpy vectorisation is outperformed by Numba, the serial Cython method does not scale very well with lattice size in this particular simulation. As expected, however, all parallel methods outperform the serial methods at larger lattices. The curves for Numba, Cython, and MPI with 4 cores flatten out more than the single-core approaches, indicating that parallelism becomes essential for keeping the runtimes low at large problem sizes.

A closer look into the performances of the parallel methods shows that Numba outperforms Cython and MPI at larger sizes. It also shows how, while MPI didn't perform as well as the other parallel methods with a lattice size of 50x50 (table 1, the message-passing overhead becomes neglibile compared to performance gains for larger sizes.

Plotting the Speedup for the parallel methods as a function of the number of workers (4) shows how all methods

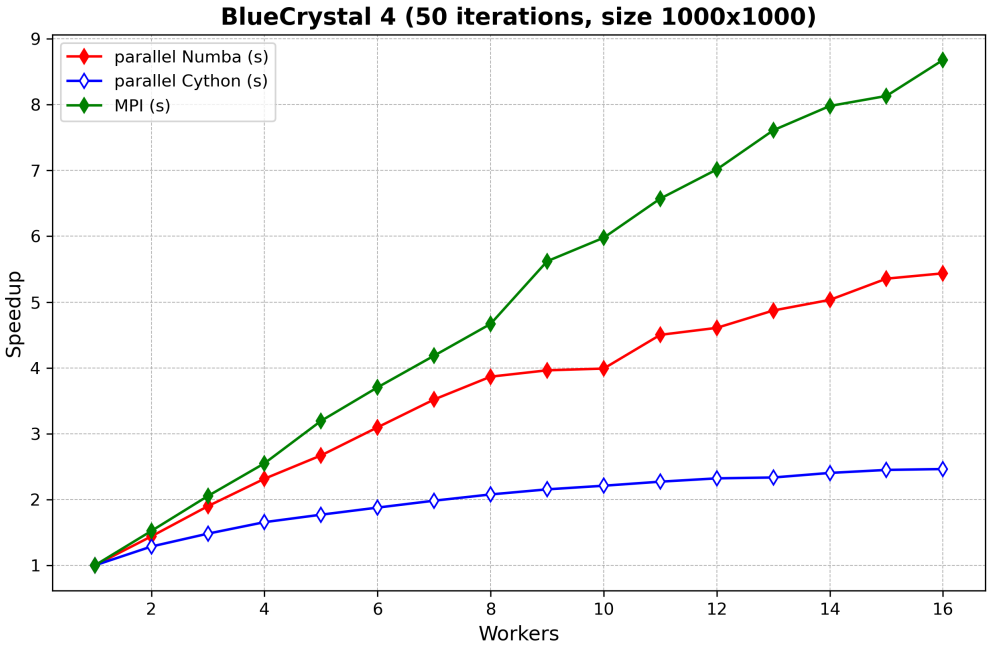

Figure 4: Speedup for the parallel methods as a function of icreasing number of workers. Each simulation was run over 50 iterations, with a lattice size of 1000x1000 and reduced temperature of 0.65.

experience a very significant reduction in runtime. It can also be seen, however, how the curves eventually come close to a plateau, having maximally exploited the parallel portions. When that happens, the serial sections that remain as well as the overhead from the parallelism limit any gains.

Figure 3 shows how MPI starts outperforming Numba above 9 workers, but both exhibit very similar curves. Cython, on the other hand, plateaus much quicker. Comparing the complexity of implementation of Numba and MPI leads to the conlcusion that the former is more beneficial - but given the nature of MPI, this can be assumed to only hold true up to a certain data size point. For extremely large problems, we can expect MPI to outperform the rest.

4 Conclusion

In conclusion, for small to medium lattice size, Numba is the easiest and most effective way of accelerating the Lebwohl Lasher simulation code. It only requires minimal changes and no additional compilation steps, yet delivers very impressive speedups and parallelisation performances.

As the problem size scales, however, MPI is a very strong contender. For smaller sizes, the message-passing overhead limits its performance, which due to the complexity of its implementation may not make this method worth it. This conclusion also applies to Cython and parallel Cython for smaller ranges of lattice sizes. In this range, NumPy vectorisation performs almost on par to Numba, which shows how powerful even this library alone can be.