Accelerating the Performance of a Python Lebwohl-Lasher Liquid Crystal Model

Nathan Truong
School of Physics, University of Bristol.
(Dated: November 5, 2025)

TARGETED FUNCTIONS FOR ACCELERATION

A profiling tool provided by default from Python was used to determine the most suitable functions to accelerate.

TABLE I: Profiling results for serial code (Size: 100, Steps: 100, $T^* = 0.200$)

ncalls	tottime	percall	cumtime	percall	function
1	0.000	0.000	15.190	15.190	`<module>`
1	0.000	0.000	14.284	14.284	`main`
3,010,000	9.509	0.000	9.509	0.000	`one_energy`
100	1.826	0.018	8.295	0.083	`MC_step`
101	0.144	0.001	3.186	0.032	`all_energy`
101	2.747	0.027	2.768	0.027	`get_order`
1	0.000	0.000	0.032	0.032	`initdat`
1	0.000	0.000	0.003	0.003	`savedat`

It was found that `one_energy()`, `all_energy()`, `get_order()`, and `MC_step()` functions were most suited for acceleration as their total time spent (`tottime`) was highest.

The `main()`'s functionality has to be retained sequentially although was modified in order to enable Message Passing Interface (`MPI`), Cython and OpenMP to function.

The `plotdat()` and `savedat()` functions could have been accelerated. However the gains from the already short total time both functions exhibited seemed extremely minimal so those functions were not modified unless another function (namely `one_energy()` changed significantly from implementing an acceleration method.

OUTLINE OF ACCELERATION METHODS

Message Passing Interface

The Message Passing Interface (`MPI`) standard was used to allow the serial version of the model to run using multiple cores. To fulfil this, each CPU core was allocated an even division of rows in the lattice, with remaining rows from an odd division

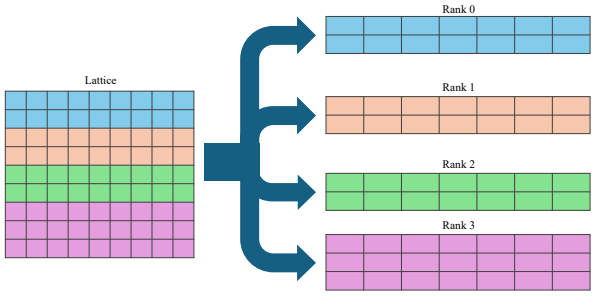

FIG. 1: Example case for 4 ranks. Each row in the lattice is split into 2 except for the last rank obtaining the remaining column that could not be evenly divided by 9.

going to the core identified as the last rank. This process was managed by the master process (rank 0) utilising the `scatter` function. Each rank then calculates its left and right neighbouring ranks, taking into account periodic boundary conditions. It also sends and obtains (`Sendrecv`) the most left and most right rows and left and right neighbouring rows of the lattice that were not received from the initial `scatter` operation respectively. These rows are used to accurately update the boundary rows during the Monte-Carlo step.

The main loop fulfilling the Monte-Carlo step and calculating the energy and order of the lattice were independently modified.

The overall energy and order parameter methods were both modified with the same philosophy. Where each rank stores a local copy of energy and order parameter and finally reduces with the sum operation and collates each term to rank 0 (`reduce(<QUANTITY>, op = MPI.SUM, root = 0)`).

The Monte Carlo step required a more intricate parallelisation strategy. To prevent multiple ranks from updating the same row simultaneously, even-indexed rows were updated first, followed by odd-indexed rows. This approach also simplified the communication of boundary rows between ranks, which occurred after each half-step, as only a sub-

set of rows needed to be exchanged depending on whether the row was even or odd. The total acceptance was also reduced with the sum operation and the values stored in each rank were collated to rank 0 in a similar manner to the method which calculated energy and order parameter. The final configuration was then reassembled using `Gatherv`. Rank 0 calculated the indices corresponding to each rank's portion of the lattice and gathered all parts to reconstruct the complete lattice, which was then used to generate the final image.

Cython

Pure `c` Conversion

In addition to creating a setup and a run file to allow this method to be developed, the main model was modified through the usage of type hints, providing `c`-definitions (`cdef`) to all variables and using the cython equivalent of `numpy` (`cnp` instead of `np`). Additionally, the `@boundscheck(False)` and `@wraparound(False)` decorators were implemented to disable index validation checks and negative indices to provide performance gains. This was implemented whenever possible and had to be omitted for some functions in the cythoned `OpenMP` and `MPI` cases.

`OpenMP` - Threaded Parallelisation

As Cython compiles Python code to `C` code, the model was able to be ported to `OpenMP`. Modifications were mainly made to `all_energy()` and `get_order()` by utilising `prange(<INT>, nogil=True, num_threads=<THREADS>)` on outer loops of the code.

`MC_step()` was left untouched, as managing the random lattice cell access (not to be confused with *random access* memory) required a more intricate setup due to Cython's implementation of OpenMP being relatively rigid.

Message Passing Interface

Combining Cython and `MPI` required the `MPI` implementation first, and then to follow the method outlined in the *Pure `c` Conversion* section.

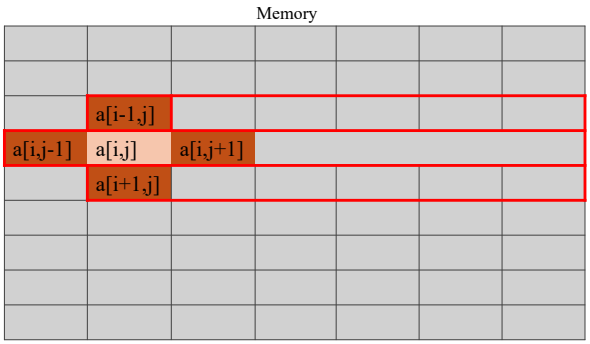

FIG. 2: Memory is accessed more efficiently in an array through sequentially accessing rows rather than at random. Any vector operation can also be quickly completed along a row/entire array.

`Numpy` Vectorisation and Memory Access Optimisation

In order to scout out opportunities for vectorising any utilised arrays, functions that heavily relied on for loops were targeted. However, two functions had to be excluded from vectorisation, `main()` as some form of sequential iteration had to be retained and the main loop and `MC_step` as sequential actions had to take place for each randomly selected site.

For the `all_energy()` function, the differences with neighbouring sites were computed using array slicing in the left, right, up, and down directions. Energy contributions were calculated simultaneously for all sites using vectorised mathematical operations as a result of using `numpy` arrays, and the total energy was obtained with a single `numpy.sum()` call.

The `get_order()` function was optimised through utilising `numpy` functions to fulfil the computation of various numerical operations needed to find the order parameter. After this, the final tensor was assembled using array indexing before it can be solved with `numpy`'s linear algebra solver. The indexing and assembly was formulated by inspection of the serial loop.

As the Monte Carlo simulation inherently relies on stochastic processes, it was not possible to fully vectorise the main simulation loop. However, to improve performance during random lattice site selection, sequential sampling was adopted. The precomputed random indices, which was already algorithmically implemented in the serial code, were ordered by row and then by column, aligning with the

row-major memory layout of `numpy` arrays.

`Numba` - Just in Time Compilation

The `@numba.jit` decorator was applied to `one_energy()`, `all_energy()`, `get_order()`. Due to other functions utilising dependancies that `numba` did not support, this decorator could not be placed on the other functions. The program was executed once to allow the just-in-time (JIT) compiler to compile the functions, thereby reducing overhead in subsequent runs.

TESTING

A visual inspection of the final lattice configurations was first used to identify any discontinuities, as shown in Figure 3, providing a quick qualitative check of implementation correctness. To verify the implementations more quantitatively, a set of pre-configured lattice states of varying sizes was generated and used as input. Each implementation was then executed 20 times at a reduced temperature of 0.2 with 250 Monte Carlo steps. The standard error of the mean for the order parameter was computed across these runs, with the percentage uncertainty required to remain below 5% to ensure statistical reliability. The order parameter was selected as, physically, it should converge to approximately the same value across all acceleration methods. A temperature of 0.2 was chosen because it lies outside the critical region, where phase transition behaviour is not expected, ensuring comparable equilibrium behaviour between implementations.

EASE OF CODING, FLEXIBILITY AND MAINTENANCE

The easiest approach was using `@numba.jit`. This required to simply put `@numba.jit` on the top of all function definitions. Although, since this only works with functions that use dependencies that `numba` knows how to compile, it is not very flexible as a result.

The most flexible approach was the pure `Cythoned` approach. By simply being able to `cdef` most variables and implement type hints, the program was able to maintain it's existing `numpy` functionality and gain performance boosts. By extension,

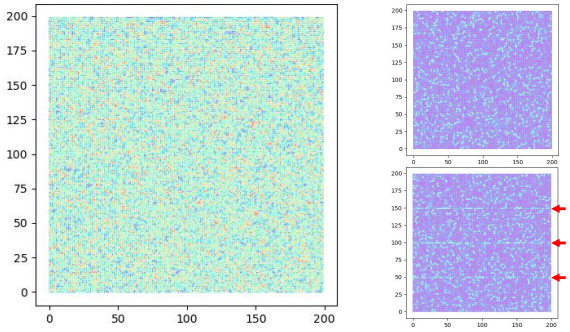

FIG. 3: Correct and incorrect `MPI` implementation comparison. The top right plot is correct. The bottom left is not as it has noticeable lines where communication between ranks was clearly not established. This is highlighted further with the red arrows.

this approach exhibits superior maintainability, as subsequent modifications to the Cython implementation can be performed in a manner analogous to conventional Python development, while adhering to the syntactic conventions of a `c`-style dialect.

The `MPI` implementation proved to be the most challenging, as managing distributed data storage and ensuring correct inter-process communication required careful coordination and substantial algorithmic modification compared to the other methods. In terms of maintainability, the `MPI` implementation is the most complex, as changes to the algorithm often require corresponding updates to the communication logic and data distribution scheme. Combining `MPI` with Cython inherits the same disadvantages associated with using pure `MPI`, while introducing an additional layer of complexity due to the need for careful type declarations to fully exploit the performance benefits of Cython.

STATISTICAL RESULTS

Figure 4 illustrates that the average order parameter over 20 iterations is consistent across all acceleration methods, with differences between methods remaining within ±2 standard errors of the mean (SEM), as evidenced by the overlapping SEM boundaries.

For single-core optimisations, `numba` showed the greatest performance gains across all lattice sizes, reducing run times by an average of 85% relative to the serial approach. By contrast, the vec-

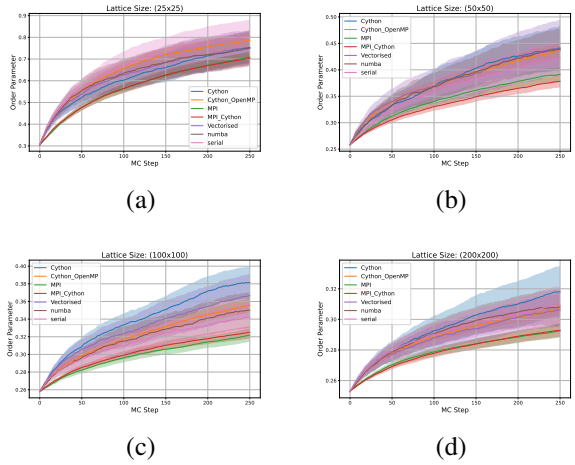

FIG. 4: Progression of order parameter for different lattice sizes over various Monte-Carlo steps.

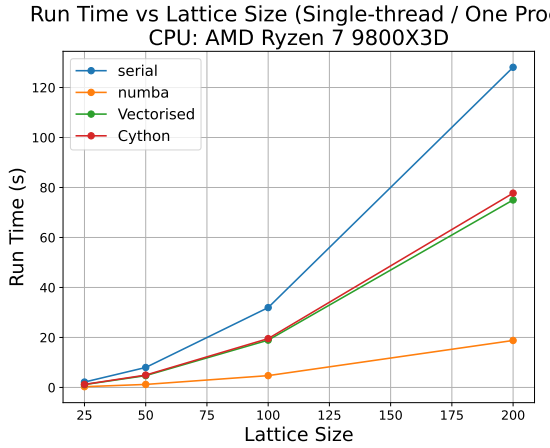

FIG. 5: Average timings for 20 iterations over different single-core accelerated computing methods.

torised and Cython optimisations offered smaller improvements of less than half the benefit provided by `numba`. This is all evident in Figure 5.

In scenarios enabling parallelism, Figure 6 demonstrates that `MPI` achieves notable speedups by utilising all cores of the AMD Ryzen 7 9800X3D, with performance gains increasing for larger lattice sizes. Interestingly, Figure 7 reveals that `OpenMP` via Cython often delivers even greater performance improvements, despite the fact that the `MC_step` function—arguably the most computationally intensive part of the code and was not parallelised in this implementation. This striking result suggests that the efficiency of `OpenMP` for the remaining

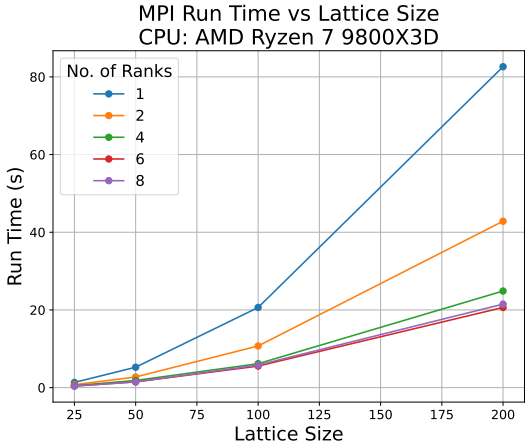

FIG. 6: Average timings for 20 iterations over `MPI`.

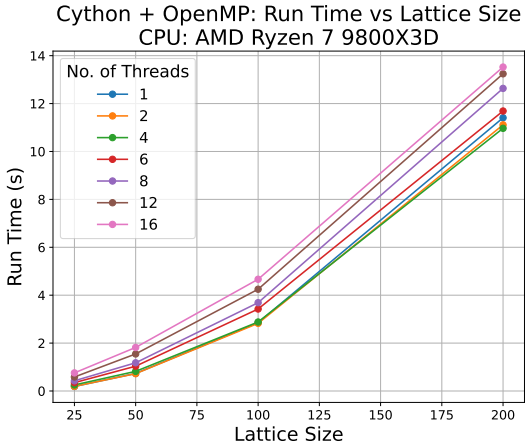

FIG. 7: Caption

parallelised sections is sufficient to outperform `MPI` which may be an indicator that the amount of overhead from `MPI` had a much larger impact in this scenario.

DISCUSSION

Putting all the best performance gains together, it can be seen that using 4 threads in OpenMP provided the best performance boosts with a percentage decrease in run time compared to serial of 90% on average. It is worth emphasising that, despite the lack of explicit parallelism, `numba`'s JIT compilation achieves performance very close to that of the parallelised `OpenMP` implementation, requiring minimal effort from the programmer.

While there may be some contradiction between

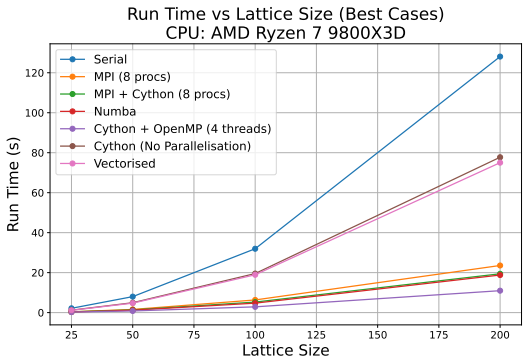

FIG. 8: Overall plot of best timings obtained from the various methods investigated

Figure 5 and Figure 7 regarding Cython timings, the implementation of `OpenMP` required C level definitions of the generated lattice and other arrays compared to `numpy c` arrays (`cnp`) otherwise segmentation faults would occur likely from race conditions.

Further investigation into how each acceleration method scales with lattice size could help determine whether `Numba` is the most effective approach, particularly when the programmer's effort is a significant factor in selecting the optimal method.

CONCLUSION

This report presented a variety of accelerated computing methods for a python Lebwohl-Lasher liquid crystal model. From an AMD Ryzen 7 9800X3D test system, it was found that using `OpenMP` with 4 threads provided the best performance gains with a 90% reduction in run time on average over the tested problem sizes. It is worth noting that applying the `@numba.jit` decorator to compile the relevant functions in the model achieved a comparable speedup of approximately 85% on average across all problem sizes, requiring the least programmer effort among all methods.

Future studies with this work could involve investigating into other `numba` decorators that could provide further performance boost. Additionally, performing simulations with much larger problem sizes with `numba` and `OpenMP` could provide additional perspective and insight into the superior method for its performance and effort to implement.

CODE AVAILABILITY

The code for this project is available on GitHub: `https://github.com/The-Great-Nate/Accelerating-the-Lebwohl-Lasher\-Model-of-Liquid-Crystals.`.