

2-D Lebwohl Lasher - Code Review

Bryn Drury

Level 3 MSci. School of Physics, University of Bristol.

(Dated: October 30, 2024)

GitHub Repository

INTRODUCTION AND OVERVIEW

This project's scope was to speed up a given piece of code while maintaining that the code could be executed as a Python script. The code provided is a Lebwohl Lasher liquid crystal simulation model in 2D. The liquid crystal is represented via an $N \times N$ dimensional grid, with the elements storing their corresponding orientation.

The energy of a given element of the grid is assessed by comparing its angle to its neighbours; the more aligned it is with them, the lower its energy. The energy of the entire system is calculated by summing over all the energies of all the cells in the system. This energy is calculated and stored for every iteration. The degree of the liquid crystal order is approximated using the order parameter. The order parameter measures the degree of orientation using a liquid crystal order tensor (found in the documentation) where the components are populated, the eigenvalues are calculated, and the largest is taken as the order parameter.

The final parameter to consider is the ratio parameter, which is calculated as a result of the metropolis Monte Carlo step (hereby referred to as the MC step or iteration). Within the MC step, a cell is randomly selected, and its energy is calculated. The orientation of the cell is deviated by an angle proportional to the system's temperature. The energy is then calculated for the new arrangement of the cell. If the new energy is lower, the change in angle is accepted; if it's not lower, a random number is generated via a uniform distribution from 0 to 1 and the Boltzmann factor is evaluated using the formula,

$$R \leq \exp\left(-\frac{E_1 - E_0}{k_B T}\right) \quad (1)$$

If this condition is not met, the change in angle is undone. This process is repeated $nmax^2$ times. The ratio of accepted to rejected changes is the ratio parameter; this is stored for each iteration.

All code for this project can be found in the git repository here. For more details on the physics behind the project, please refer to the project specification listed as AccelCompMiniProj2024.pdf in the GitHub repository.

METHODOLOGY

Original Code

Before discussing any optimisation, it's best to consider the original code. After that, when considering different implementations, if any code, function or implementation is not explicitly mentioned, it can assumed to be effectively identical to the original code. The code that completes the computation is implemented entirely in Python using a combination of pure Python objects and the NumPy library. The code also uses matplotlib to show the system's state before and after the computation.

The functions `plotdat` and `savedat` remain unchanged for all implementations henceforth and are not timed, so they should not influence the programs and computation runtime. The `plotdat` function produces a quiver plot of the crystalline structure; there are three options: the first will colour each cell's quiver by its energy, the second colours them by the angle, and the third has no colour. The `savedat` function saves the inputted parameters and the parameters of each simulation step to a text file with the time in the name.

The `initdat` function is also run outside the timed section of the code, so this does not affect execution time. This function initialises the 2-dimensional array populated with random numbers. This array is the representation of the crystal. This code is implemented using the NumPy random module to generate the numbers between 0 and 1; then, they are all multiplied by 2π to produce a full range of orientations.

As stated before, the `one_energy` function takes a coordinate input and calculates the energy of a given cell by considering its angle related to its neighbours; in this implementation, this is done simply by increasing an energy value for each of its four neighbours, accounting for wrap-around at the grid boundary. The wrap-around is such that a cell on the left edge ($x = 0$) will consider its left neighbour as the cell on the right edge ($x = nmax$) with the same y value.

The `all_energy` function loops over all the cells in the system, calculating the energy of each using the `one_energy` function and summing all the energies into a single value. This summed energy is what the function returns.

The `get_order` function creates the aforementioned tensor, which consists of a three-layer stack of the sin of the orientation values of the crystal array, the cosine of the values and then the The function returns the order parameter of the current crystal array arrangement.

The `MC_step` function generates three random $nmax$ by $nmax$ NumPy arrays, two called `xran` and `yran`, which are

populated with random integers ranging from 0 to n_{max} and are random index positions and the last is called *aran*, which is populated with values from 0 to $0.1 + T$, where T is the temperature argument passed when running. The function then loops through all possible combinations of i and j from 0 to n_{max} and selects an element of the crystal array based on the integers stored in the (i, j) positions of the *xran* and the *gran* arrays. This random selection over $n_{max} \times n_{max}$ loops means that the program will, on average, check every cell once per iteration. It then calls the *one_energy* function to calculate the energy of the selected cell; it is perturbed by the amount found at the index (i, j) of the *aran* function. The rest of the function follows the description in the introduction, whereby it perturbs the angle, compares the energy to its previous state, and accounts for the Boltzmann factor where needed. The function returns the ratio or acceptance rate of the iteration.

The main function for this code calls the *initdat* function, initialises the arrays for storing the energy, ratio and order values for each iteration, and populates the first position with the corresponding values from the initial conditions. The program starts a timer and enters the iteration loop. This section calls the *MC_step* and stores the returned value in its corresponding iteration location in the ratio array. Next, it calls the *all_energy* function and stores the return values in the energy array. Then, it calls the *get_order* function and stores the returned value in the corresponding location of the order array. The timer is then stopped, the runtime calculated, and the parameters and final results of the simulation are outputted to the command line. The program saves the data stored in the ratio, energy, and order arrays by calling the *savedat* function. Then, it plots the final arrangement of the crystal array using the *plotdat* function. Profiling this code using the *cProfile* library provides information about the execution of the program. An example of which can be found in the GitHub Repository. It shows that the most time-consuming methods are the *MC_step* and the *get_order* functions. These large cumulative time values are expected as they are the most complex and are called many times. The large number of calls and time per cell indicated that any effort to optimise would be most beneficial here.

NumPy

This implementation tries to use as many NumPy functions, including using vectorisation where possible. The *one_energy* now uses a vectorisation by using NumPy's array indexing to calculate the difference in angle. The *all_energy* function now uses the NumPy *roll* function to shift the array such that all the neighbours can be calculated at once by subtracting them from the original array, removing the need for a loop. The *get_order* function now uses NumPy's *einsum* function to calculate to remove the need for a loop. However, the *linalg* module is retained to compute the eigenvalues. The

new *MC_step* now uses two simpler loops to calculate the two energies for all the cells and stores them in two 2-dimensional arrays. It then creates a 'delta' array by calculating the difference between these energy arrays. It then constructs an acceptance 'mask' by comparing whether the delta array is smaller than or equal to zero (i.e. if the step is energetically favourable). It also calculates the Boltzmann factor and random number where needed. The accept value is then calculated by taking the sum of the acceptance mask and dividing it by the square of the array length (like the original code). This method abstracts much of the calculation into the NumPy library modules. Although it does use double the number of loops now, the abstraction increases the performance enough that this was a non-issue.

Numba

This was the simplest to implement, using the previously described NumPy implementation and adding the just-in-time (*jit*) decorator in front of the *one_energy* function declaration. Nothing else was changed, as this function is used heavily in the *MC_step* function, and this change significantly improved.

Parallel Numba

To multithread the Numba implementation, it was required that the *MC_step* function was compiled as well. To implement this compilation and parallelisation, the *jit* decorator was added in front of the function declaration with the parallel condition set to true. Numba, however, disliked using NumPy's advanced indexing to apply and revert the changes to the angle values, so these were replaced with *prange* for loops. The other loops in the *MC_step* function were also changed to *prange* loops. No other changes were made to the single-threaded Numba implementation.

Cython

I took the original code and made the necessary changes to run it as Cython code for this implementation. Essentially, all that was changed was using data type in the declarations such that the code is now statically typed and then compiled. I created a separate Python script to import and run the Cython functions, but no other changes were made. The plot function was kept in standard Python.

Parallel Cython

This is the same as the previous implementation of Cython but with a *prange* on the outer for loops in almost every location. The only one that did not get this is the small loop

in MC_step, which reduced the performance. Some imports were added, and some changes were made to the setup file, but these were just adding the OpenMP compilation flags.

MPI with pure Python

This implementation uses a modified version of the original code. It uses the mpi4py library to communicate between multiple processes. Each process follows the standard flow of the program as specified in the original code; however, within the all_energy and MC_step functions, the work is distributed across multiple processes. The root process generated the initial array and distributed it to the other workers. The communication in the all_energy function only requires that the root thread has the result such that it has the information to save at the end of the program's execution. However, the MC_step is working on updating the state of the grid array, meaning that not only do the results need to be combined, but they also need to be redistributed back out to all the processes such that they have the required information for the next step of the simulation. No changes were made to the get_order calculation, and the implementation for distributing the work was deemed too complex for the expected speedup. Also, no changes were made to the initdat, savedat, and plotdat functions other than an early exit for the none-root processes.

MPI with Parallel Cython

This implementation used the same logic as the pure Python MPI implementation but with the parallel multithreaded functions in the Cython implementation, with the same minor changes such that the work is distributed and communicated between each process. So, this implementation allows for computation across multiple nodes, each with the ability to take advantage of multiple threads.

C++ Python API

This uses the C/C++ Python API, designed to write Python libraries. This means that I could port the C++ code by simply creating an interface using the API library provided by the Python Software Foundation. In this implementation, I ported both the main function and the individual functions such that the performance could be compared between running the code entirely via C++ (calling the main function) and running the simulation in Python by calling the MC_step, get_order, and all_energy functions for each iteration.

C++ Benchmark

A good benchmark was needed to accurately compare the speed of all the approaches. So, for this implementation, the

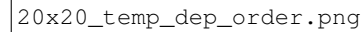


FIG. 1. Caption

code follows as closely as the original code as possible, but with the use of OpenMP to multithread operations where possible.

RESULTS AND DISCUSSION

Each program was run on a MacBook Pro with an M3 Pro CPU to analyse and compare the results. This CPU has twelve cores in a big little configuration with six performance and six efficiency cores. The CPU is based on a consumer ARM architecture, meaning that it has 64-bit wide registers. Each program was run on a MacBook Pro with an M3 Pro CPU to analyse and compare the results. This CPU has twelve cores in a big little configuration with six performance and six efficiency cores. The CPU is based on a consumer ARM architecture, meaning that it has 64-bit wide registers. As part of our validation process, we recreated the graph shown in Figure 1 from the task specification. This graph compares the order parameter at the end of the simulation to the reduced temperature of the system. The results, as expected, confirm that the order of the system at the end of the simulation decreases at a reduced temperature value of $T^* = 0.9$.

As the task test is about the performance of the simulation, given that the results are the same, it was essential to understand the runtime dynamics for the given input parameter. There are two factors to consider. The first is the number of iterations, and the second is the grid size. The number of iterations is expected to yield a linear relation to the program's runtime. This relation was observed and can be seen in the Figure 2, where the relationship is strongly linear. The second factor is the grid size, where given that the grid is square and dependent on the input parameter $nmax$, an increase in $nmax$ is expected to result in a squared relationship on the runtime. Given this, it would be expected that graphing the runtime against the $nmax$ parameter on a logarithmic scale will result in a linear relationship.

As the number of iterations yields a linear relationship to the runtime, and there is no need for the model to reach its

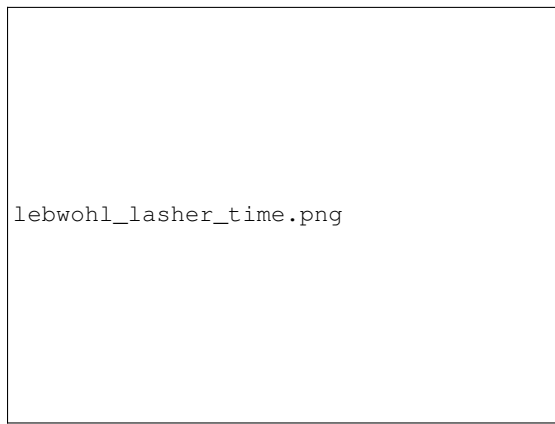


FIG. 2. Caption

equilibrium, the value settled on was fifty iterations. This number of iterations ensured that the main loop was the driving factor in the runtime, and for the slower approaches, the runtime was not too short or excessively long. Each implementation was tested with various values for n_{max} until the runtime exceeded three hundred seconds. The implementations will be analysed in three groups:

1. the Python implementations (the original, the NumPy version, and the MPI version)
2. the compiled single-threaded implementations (the single-threaded Cython and Numba versions)
3. the compiled parallel approaches (the multi-threaded Cython, the multi-threaded MPI implementation, and the C++ Python API version)

The pure C++ code will be a performance benchmark for the single and multi-threaded implementations, as the number of threads it uses can be specified at runtime via command line inputs. The report did not include these, but the code was left available on the GitHub repository.

Pure Python Implementations

The graph in Figure 3 shows the speed-up over the original code achieved for the NumPy and the MPI implementations using varying grid sizes. The data for NumPy shows that this implementation is consistently about 1.5 times faster than the original. This is probably because some functions take advantage of vectorisation, filling the registers more efficiently. As the registers on the computer used are just 64 bits, and the numbers used are floats, which take up 32 bits, the calculation for two numbers can be done at one time. An exact two-time performance increase is not seen because this optimisation is only possible for a portion of the code. The other factor is that the backend of a lot of the NumPy library is written in the C language, which is compiled and will run faster than the native Python version. The other curve seen in



FIG. 3. Caption

the plot for the MPI implementation demonstrates a 2.25 times speed-up for the code using 6 MPI processes. This is a disappointing result, given that the work is being processed in parallel across six processes, indicating a significant overhead for the communication between processes. This implementation would allow the computation of larger systems as the memory requirements are distributed across the process, all of which can be computed on different nodes of a supercomputer or network.

Single-Threaded Compiled Implementations

The graph shown in Figure 4 shows the speed-up achieved by using compilation techniques for Python. The first is Numba; as mentioned, this compiles a just-in-time compilation before running the program, then caches the compiled program for later use. This approach saw a significant speed-up for larger grid sizes but not smaller sizes. This is probably due to a combination of factors that occur in the process. It is important to note that the program was run multiple times before the given time was produced to ensure the function was compiled, cached, and ready for greater speed-up. This issue is that uncaching the functions and other optimisations that Numba might do will have a more significant overhead that is only overshadowed when using larger grid sizes. This implementation shows its strengths at larger sizes, achieving over a ten-times performance increase over the original code. The other data on the graph shows the speed-up achieved using the single-threaded Cython implementation; this implementation almost matched the performance of the numba implementation but without the lacklustre performance at smaller grid sizes. Given that this implementation is meant to convert the code into C, this result being as slow as this, even slower than the Numba implementation, is somewhat unexpected. This indicates that the code has to convert between the Python objects and the C



FIG. 4. Caption



FIG. 5. Caption

type too frequently, creating too much overhead and reducing the performance potential of the program. This implementation achieved a modest speed-up of approximately nine times over the original code.

Multi-Threaded Compiled Implementations

Moving onto the multi-threaded results, the performance improvements were far more significant. The graph is shown in Figure 5 shows the speed-up of these implementations. These implementations experienced minor performance increases at small grid sizes; the Numba implementation increased the program's runtime as the smallest grid size tested. The reason for all of these is that the overhead of distributing the workload is comparatively more significant when the computation is smaller for smaller grid sizes. This observation is backed up by the fact that all implementations level out at high grid sizes. The slowest of the implementations on this graph was the Numba version; this slowdown is probably caused because most of this code is still written in pure Python, and it is only the `one_energy` and `MC_step` functions that are implementing Numba's jit and only the `MC_step` function that takes advantage of multi-threading. This implementation achieved a peak speed-up of 80 times faster than the original. The subsequent implementation was the multi-threaded Cython version. This code used oranges to parallelise as many for loops as possible, allowing it to obtain significant performance improvements, achieving a maximum speed-up of 277 times faster than the original. The following implementation was the MPI version. However, the version also uses almost the same multi-threading as in the Cython implementation; it is also heavily reliant on the use of the MPI system, meaning data communication is a limiting factor. Again, as in the pure Python MPI implementation, the program can calculate the larger systems where the memory requirements are

distributed over multiple systems. This implementation achieved a 148 times speed-up over the original code. The final implementation is the *crème de la crème*, the finest of them all, the way god intended code to be written, in C++. This implementation uses the Python API, which makes the C++ function available to call from C++; there are two options for running this version: the conventional method, whereby all the functions are called in the main Python script, mimicking what the other implementations do. The other option is the abstracted version, which calls on a main function written in C++ such that all the execution takes place in compiled code and returns the relevant information for plotting. This later version is the one used in testing and provided the most significant performance increase of all the implementations used, giving a speed-up of over 513 times the runtime of the original code, almost double the parallel Cython code, which was the next fastest implementation.

DISCUSSION

Each implementation will be evaluated by performance improvement and the time and effort taken to implement it. When writing the code, I attempted to maintain a consistent amount of time for each so that this evaluation would be fair. The reason for using Python is its versatility and ease of use; its design as a language inherently sacrifices performance and, therefore, heavily relies on external libraries to improve performance. This sacrifice is okay if the amount of time and effort spent on using the libraries is less than it would have been to use another, more performant language. The first to discuss is the NumPy implementation; although this took a while, as the initial program had to be decoded and understood how to use the NumPy functions such as `einsum`, this took a fair amount of time and effort. However, the performance improvement that this version yielded was somewhat disappointing; It was expected that heavier use of

the NumPy function would have more of a performance improvement. This implementation is only really worth it if one is already well acquainted with the code and the NumPy functions, such that they can be used during the initial development. Next are the Numba implementations; both the single and parallel approaches were straightforward to implement, required very little time, and provided highly respectable performance improvements. The Cython versions were highly time-consuming to implement, as Cython is very prescriptive about where one can and cannot put Python variable types. The performance increase of this implementation was also less than expected, given that the code is being compiled into C/C++; this is probably because the code still uses the NumPy library rather than C types the array, meaning there is some overhead on the operations. These would be a manageable change but require more code refactoring, but enough time had already been spent on this task, well over the amount of time initially set out for these versions. In the end, the parallel Cython version was the second fastest. Next, the MPI code versions gave reasonable performance uplifts that were not proportional to the time used. However, the reason for using and implementing MPI is for the computations of larger systems that would exceed the memory requirements of a single system. For this reason, using and implementing MPI can be worthwhile, depending on the case. The most performance, documented and widely used implementation was the C++ Python API. The abundant resources and guides provided for using the API make the development process extraordinarily streamlined, and given that no other implementation reached anywhere near the speed up it achieved, it is the apparent choice for high-performance Python code.

CONCLUSION

Using NumPy provided a 1.5 times speed-up, which made it not worth the time and effort to implement, given the other options available. The two Numba implementations were effortless and provided a healthy performance improvement. They also provided an 11- and 80-time speed-up for the single and multi-threaded implementations, respectively, making them well worth implementing. The Cython implementations took significantly longer than any other implementation to develop and debug, but they did provide a 9 and 277-time speed-up of the single and multi-threaded implementation, respectively. If one is well acquainted with producing Cython code, this is a valid option for speeding up code. The pure Python implementation of the MPI code was not worth the time; the performance improvements were mediocre at just 2.25 times speed-up for the time to develop. However, the Cythonised version of the MPI code produced a healthy performance increase of 148 times faster. It would allow the computed systems to reach larger sizes than other implementations. Finally, the C++ Python API provided the most significant performance increase at over 500 times

speed-up. Although already acquainted with C++ and Python, using the API was novel, but the documentation and guides made the development fast and well worth the effort.