Accelerating the Metropolis Algorithm for the Lebwohl-Lasher Model

I. Setting of the Problem

The Lebwohl-Lasher model simulates the behaviour of nematic liquid crystals (in their "nematic" phase, the constituent molecules have no positional order but tend to point in the same direction over long-range domains. It partitions space into a lattice and assigns an average orientation to the molecules within every cell. Working in 2D, this will be a single angle θ_i known as the liquid crystal director. The potential energy associated to a single director depends on its 4 nearest neighbours and is proportional to the 2$^{\text{nd}}$ Legendre Polynomial of the cosine of the relative angle between said cells:

$$U_i =: \epsilon \sum_{\forall \langle ij \rangle} \frac{1}{2}(3cos^2(\theta_j - \theta_i) - 1)$$

It is customary to compute energies in "reduced" units of ϵ, the maximum interaction energy between any 2 liquid crystal directors. The dynamics of the system as it thermalizes at temperature $T^* =: \frac{k_B T}{\epsilon}$ are simulated with a "Metropolis-Hastings Algorithm". In essence, for a number of Monte Carlo steps one proposes a random (but normally distributed s.t. $\sigma = 0.1 + T^*$) deviation to every director. If the potential energy of the updated director is lower than before, the proposal is automatically accepted. Else, the proposal is accepted with probability $e^{-\frac{U_{new} - U_{old}}{T^*}}$ - so a director can increase its energy as a result of thermal fluctuation. The main bottleneck of the algorithm is the cost of each Monte Carlo step, as due to the non-locality of the potential energy each director must be updated one by one - consider that a random deviation to a director would affect the potential energy on its own site as well as its 4 neighbours. This would imply the runtime per MC step scales as $\mathcal{O}(N^2)$ where N is the lattice dimension. However, as we shall see there are several strategies to accelerate the Metropolis Algorithm owing to the fact that interactions are contained to nearest neighbours.

II. Vectorization

Whereas a list could contain objects of a variety of data types, an array can only contain data of a single type. The homogeneity of an array's elements in data type means NumPy is able to delegate the job of performing mathematical operations on the array to efficient, compiled C code. As such, one expects to see large speedups when cleaning out loops in favour of vectorized NumPy functions.

In order to avoid the nested spatial loop in each MC step, one must first alter the algorithm to allow for a block of sites to be updated simultaneously and independently from one another. One such solution is to use the "Checkerboard Decomposition" - a technique common to simulations of the Ising Model. We split the lattice into a checkerboard of "black and white" tiles; to be exact, we group sites by matching/mismatching row and column parity. A black tile only has white neighbours and vice versa; therefore, its director can be updated independently from all the other black tiles since none of their interaction domains overlap. As such, it is possible to vectorize the entire Monte Carlo step to update the full block of "black" tiles, followed by the full block of "white" tiles. The non-locality is restored in that you must first update one block and then its complement, although a loop of length 2 is far better than a nested loop of length N^2. We implement this "checkerboard" vectorization in all 3 of our strategies for accelerating the Metropolis algorithm, as it is what allows parallelization in the 1$^{\text{st}}$ place!

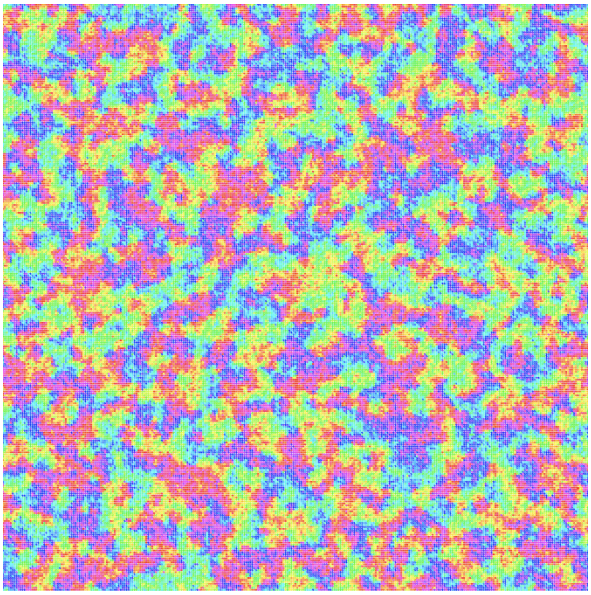

Fig. 1: Angle-coloured heatmap of a 1000×1000 lattice after 50 Monte Carlo Steps for $T^* = 0.5$, showing how long-range domains of aligned directors self-assemble in the nematic phase.

III. MPI

Our first parallelization strategy uses MPI to portion the computation of each vectorized MC step across NP parallel processes. The master rank initialises the lattice of directors,

and distributes chunks of rows across each worker as evenly as possible. On the worker process enough memory has been set aside to handle that chunk of rows and no more - besides 2 additional rows above & below where the interaction domains of the uppermost and lowermost rows overlap with their converse extremes in the neighbouring chunk (N.B. the master rank also sent messages to each rank informing them of their neighbouring ranks above and below in the column).

The worker process then runs a vectorized Monte Carlo step on one group of the checkerboard (taking care to ensure the tiling is consistent between the sub-blocks of different worker ranks and the complete block), communicates its updated bordering rows with its neighbouring workers, and repeats this process for the complementary group. After all iterations are complete, each worker sends back its segment of rows to the master rank, which recombines them into a single final array.

This strategy is not particularly simple to code, as it requires you to construct a consistent and efficient message passing system between ranks, which can quickly become confusing when some messages are non-blocking and others aren't. However, by nature this strategy is extremely scalable with the number of processes (NP). Fig.2 shows how the simulation runtime for a variety of lattice sizes is affected by increasing NP:

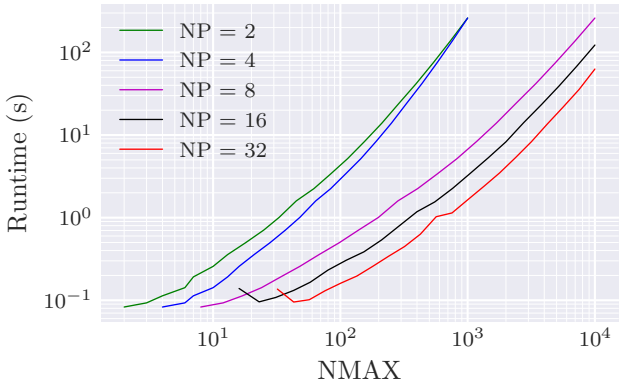

Fig. 2: Simulation runtime (s) versus lattice dimension NMAX for MPI strategy on a log-log scale, where different lines represent different choices of NP. Each simulation was run for 50 MC steps at $T^* = 0.5$ s.t. the final directors were clearly assembled into domains.

At 2 and 4 processes, the scaling with NMAX is near quadratic so it would seem the overhead of message passing and maintaining the COMM_WORLD outweighs the advantages of parallelisation for low NP. However, on multiplying to 8, 16 and 32 processes we reap the benefits of MPI: notably the gradient is far shallower at high NP, implying the scaling with NMAX is closer to linear; in addition, there appears to be a roughly constant offset between the 3 lines, which on a logarithmic axis implies the runtime is shrunk by the same ratio each time NP is doubled - an indication that this MPI strategy does indeed scale well with NP.

IV. Numba

Our 2nd strategy uses Numba in concurrence with OpenMP to accelerate the Metropolis algorithm. In comparison to MPI this is far easier to code, as once you have a vectorized function for MC-stepping most of what remains is including the necessary *@jit* decorators. Rather than portioning out chunks of rows to each process, we use a parallel for loop over rows to compute their potential energies after each update - each row is computed with vectorized operations. Fig.3 shows how the simulation runtime for a variety of lattice sizes changes with increasing number of threads:

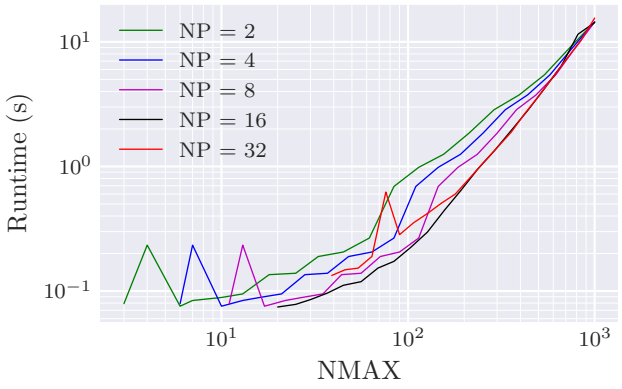

Fig. 3: Simulation runtime (s) versus lattice dimension NMAX for Numba+OpenMP strategy on a log-log scale, where different lines represent different choices of NP. Each simulation was run for 50 MC steps at $T^* = 0.5$.

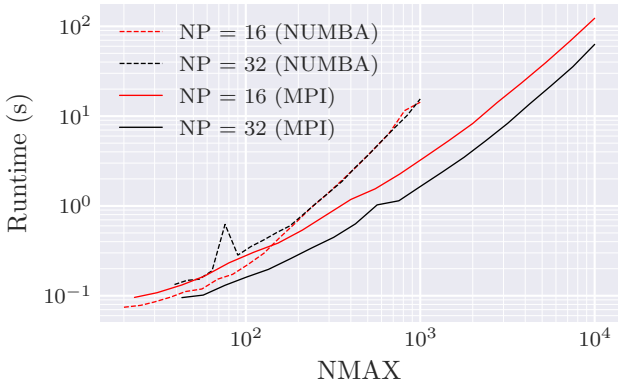

Fig. 4: Comparison between simulation runtime for MPI and Numba+OpenMP strategies over a range of lattice dimensions NMAX.

Evidently there is not much advantage in increasing the number of parallel processes, as unlike for MPI at high NP we see in Fig.3 that the lines converge together as NMAX increases. Moreover, from Fig.4 we compare the runtime scaling between our MPI and Numba+OpenMP

strategies: at small lattice sizes the runtimes are similar, although they quickly diverge - the MPI strategy scaling closer to linearly and the Numba+OpenMP strategy scaling quadratically.

V. Cython

Our last strategy attempts to "cythonize" the metropolis algorithm. The parallelization approach is the same as with Numba and simply consists of using parallel for loops over rows. Beyond that, Cython mostly just requires you to declare explicit variable types. However, the apparent simplicity of a Cython strategy is quickly exposed when it comes to preparing the *setup.py* script for building cythonized python code on Mac OS. Furthermore, it is quite opaque as to how to use Cython on BlueCrystal. This limits the flexibility of this approach, as without BlueCrystal one is limited to as many threads as your local device contains. However, despite the limits on NP, Fig.5 demonstrates that doubling the number of threads does indeed reduce runtime by a significant ratio each time. Also, the scaling with NMAX appears to be roughly quadratic unlike with the MPI strategy:

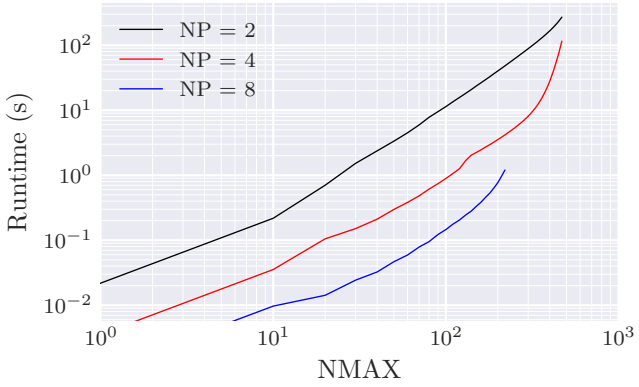

Fig. 5: Simulation runtime (s) versus lattice dimension NMAX for Cython+OpenMP strategy on a log-log scale, where different lines represent different choices of NP. Each simulation was run for 50 MC steps at $T^* = 0.5$.

VI. Conclusion

Overall, this student draws the fairly clear conclusion that an MPI strategy is superior. Despite the added difficulty of coding with MPI compared to Numba and Cython, its acceleration of the Metropolis algorithm exceeds either one. Whereas the Numba and Cython strategies scale roughly quadratically with NMAX, at high NP (NP $\geq$ 8) the MPI strategy scales far closer to linearly, which would be the theoretical ideal. Furthermore, the speedup of the MPI strategy continues to increase upon scaling NP, whereas the advantages of the other 2 approaches diminish due to the rising cost of maintaining that many threads.

VII. Github Repository

https://github.com/Joel-Oscar-Mills/Lebwohl-Lasher-.git