Lebwohl-Lasher Script Acceleration

Thomas Smith
github.com/do22555/LebwohlLasher

Common test parameters
Single Thread: **50, 50, 0.5, 0**
Multithreaded: **512, 512, 0.5, 0**

1. Base Benchmark

These scripts were run on a Windows 11 machine running WSL, with a 12th Generation Intel Core i7-1260p mobile laptop CPU with 4 hyperthreaded performance-cores and 8 efficiency-cores (12 cores, 16 threads in total). Its base clockspeed is 2.10 GHz with a seldom-reached maximum turbo clockspeed of 4 GHz. The system has a 10.8GB allocation of useable DDR4 RAM. It does not have a dedicated GPU.

The common test parameters were initially set at 50, 50, meaning the lattice dimension, L, the number of Monte Carlo Steps, N, respectively. These were set to give a runtime on the order of seconds. Later in the report, the overheads created by establishing multithreaded code dominates the overall runtime. Consequently, these numbers will be increased by an order of magnitude.

Modification of the initial script (now called `LebwohlLasher_raw.py`) was required in order to avoid a runtime error (*line 275*, `LebwohlLasher.py`). Another trivial modification to change the location of the output files was made. With this script, a benchmark mean run-time of **2.999688 s** was achieved with 5 runs using the common test parameters.

2. Script Utilisation: `LebwohlLasher.py`

Looking at the cProfile output, the main bottlenecks are:

1. `one_energy` (called 377,500 times, 2.081 s cumulative)
2. `MC_step` (called 50 times, 1.864 s cumulative)
3. `get_order` (called 51 times, 0.749 s cumulative)
4. `all_energy` (called 51 times, 0.699 s cumulative)

From this, it is evident the main contributor to runtime is `one_energy` *(line 131)*.

The time complexity of each cell in one_energy is O(1). In this simulation, one update attempt per lattice site is made per Monte Carlo step, and `one_energy` is called twice per site in MC_step *(line 210)*. Consequently, `one_energy`'s contribution to the total runtime scales as $O(N \times L^2)$, where N is the number of MC steps and L^2 is the number of lattice sites.

The cProfile output shows 377,500 calls to `one_energy`, which matches the sum of: 50×50 sites × 50 MC steps × 2 calls per site (250,000), plus 51 calls to `all_energy` in the loop (127,500 additional calls). All other bottlenecks scale with O(N).

In conclusion, the code in `LebwohlLasher_raw.py` relies too heavily on iteration where SIMD instructions are possible.

2. NumPy Vectorisation: ` '_vectorised.py`

Principally, the use of `NumPy` modules allows for the replacement of nested iteration with `NumPy` vector operations, which run in compiled C and use SIMD instructions.

The function `total_energy` *(line 40)* complemented the largest bottleneck, one_energy. This replaced neighbour lookups and loops with `np.roll,` which wraps around more efficiently. All arithmetic became element-wise across the full array. Summing with `np.sum` gives the total energy. This maintains the same time complexity $O(NL^2)$ but NumPy performs ~10^6 operations in C instead of 10^6 Python calls, reducing the overall runtime by a constant factor.

The order parameter, `get_order` was also vectorised *(line 55)*. The use of nested loops was eliminated and replaced by matrix algebra using Einstein summation (`np.einsum`) to compute the Q-tensor in C. Mathematically, the Q-tensor is a symmetric, second-rank tensor that quantifies the deviation of the system from isotropy:

$$Q_{ab} = \frac{1}{2L^2} \sum_{ij} 3l_{aij} l_{bij} - \delta_{ab}$$

Where $l_{aij} = (\sin\theta_{ij}, \cos\theta_{ij}, 0)$, the component of each molecular director and δ_{ab} is the Kronecker delta. The subtraction of δ_{ab} removes the isotropic part (the contribution which would arise if all orientations were random) ensuring Q=0 for a completely disordered configuration. The division by $2L^2$, the lattice dimension returns the average energy of the system. This tensor-algebraic reformulation is physically identical to the scalar computation in the original code.

`all_energy` was completely removed and replaced with a reference to `total_energy` *(referenced in lines 176, 183).*
`MC_step` *(line 146)* remained largely sequential due to each site's dependence on the current lattice state. Many python loops within the function were replaced with vectorised RNG and arithmetic for a minor performance boost.

The updated cProfile output gives the following bottlenecks:

1. `one_energy` (called 250,000 times, 1.4 s cumulative, ~ 30% improvement). The large volume of calls is because MC_step is not vectorised.
2. `MC_step` (still called 50 times, 1.8 s cumulative).
3. `get_order` (runtime reduced by an order of magnitude)

With this system, `all_energy` is essentially eliminated, one_energy is called fewer times and `get_order` is fully vectorised. Consequently, the mean runtime from 5 runs reduced to **0.989825 s**, an improvement of almost 70%.

`MC_step` still dominates runtime, with its cumulative time reducing only slightly, from circa 1.9 s to circa 1.8 s. `MC_step` cannot be vectorised without changing the update scheme to one updating all lattice sites simultaneously, which is likely to affect the physics of the simulation.

3. Numba: ` `_numba.py`

By compiling `MC_step` *(line 154, now* `MC_step_numba`*)*, with Numba's `@njit`, the 50 necessary calls became inline. This removed the python interpreter overhead for 250,000 iterations (with standard parameters).

`one_energy` was changed to `local_delta_energy` *(line 115)* which directly computed ΔE, avoiding two calls to one_energy. In theory, this reduced per-site computation time by circa 50%, though this is a small contribution to total runtime. `local_delta_energy` had to be modified to avoid numba-incompatible python function calls inside the JIT region.

Matplotlib is now imported lazily, cutting start time when the plot flag is zero. `total_energy` and get_order remained unchanged from the previous section, as they were sufficiently efficient in `NumPy`. JIT cannot handle file I/O or `MatPlotLib`, so these functions were also left in python.

The mean runtime from 5 runs was reduced from circa 3 s to **0.609752 s** excluding compilation time - an improvement of 80% from the original.
Other than compilation, cProfiler showed no obvious bottlenecks with this code.

4. Multithreaded Numba: ` `_numba_p.py`

Multithreading was introduced to this version. Running on 4 threads (equal to the number of performance *cores)* the mean runtime achieved was **0.024596 s.** This did not change running on 8 threads (equal to the number of performance *threads* – one would expect the number to reduce, however it is doubtful that Windows allocated all 8 performance threads to Python in this instance). Runtime did reduce subsequently on 12 threads (the total number of performance and energy-saving threads on the system), achieving a time of **0.021125 s** excluding compilation time, a reduction by two orders of magnitude from the original (Fig 1). Threads count is configured on *line 186*.

This was achieved by replacing `MC_step`_numba with a parallelised version *(line 139)*. This takes advantage of Numba's `prange` (a drop-in replacement for range, compatible with parallel processing. Chequerboard decomposition ensured thread-safe updates. By common convention these cells were defined with the sums of their integer coordinates (i+j parity); labelled "red" (even parity) and "black" (odd parity). All "red" sites were updated on the first pass, with all "black" sites updated next. This ensured that no interactions between neighbours took place on the same pass. To avoid problems occurring with a shared accumulator (multiple threads update different cells simultaneously) a per-thread accumulator was used. Parallelising MC_step in this way achieved a performance boost of an order of magnitude whether parallel processing was used or not (ie. the chequerboard decomposition alone achieved a massive performance boost).

cProfiler showed no obvious bottlenecks with this code other than compilation.

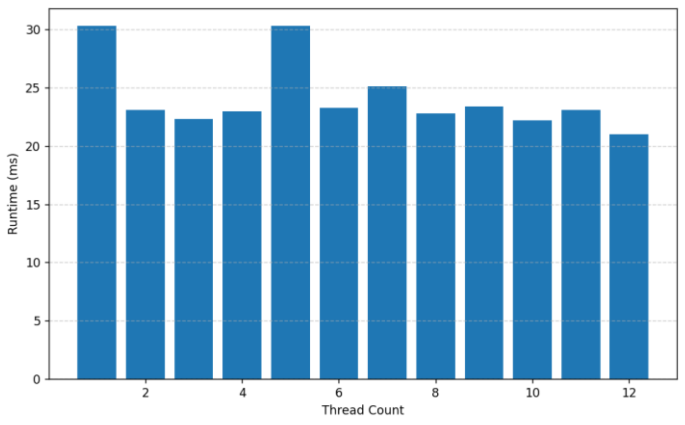


Fig 1: Numba Parallel Performance: Thread Count versus Runtime (ms). Diminishing returns begin at 2 threads. (figures.py). This could be due to this system's energy saving cores.

For both Numba code versions, compilation time sat at **2 – 3** seconds. JIT compilation is inefficient, though at larger numbers of MC steps, the contribution of compilation time is minimised as it is a constant factor in time complexity.

Consequently, for future scripts, L and N will be increased by an order of magnitude, and this is reflected in updated common test parameters.

5. Mid-Point Sanity Check

A script was written (`./mid-point/plot.py`) to display the output data of a single run of each of the four scripts:

- Simple Python: `LebwohlLasher.py`
- Vectorised: ` ' '_vectorised.py`
- Series Numba: ` ' '_numba.py`
- Parallel Numba: ` ' '_numba_p.py`

The physics during runtime appeared consistent, however the order parameter initialised incorrectly in all three optimised programs. This suggested an issue with `get_order`'s initialisation.

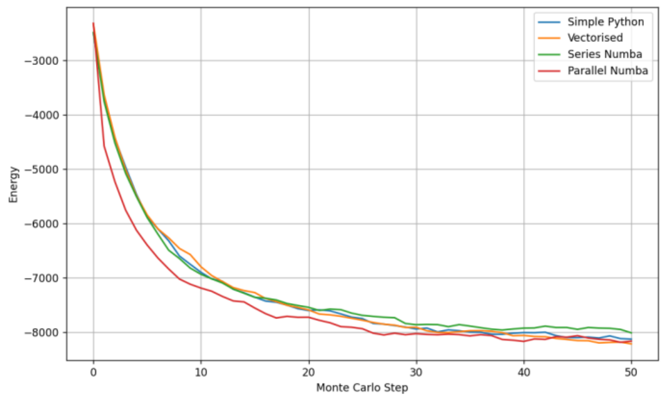


Fig. 2: All four methods showed identical energy progressions, ie. the physics had not changed with optimisation.

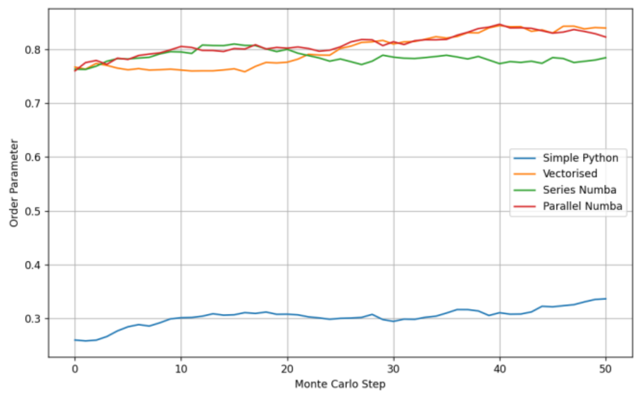


Fig. 3: While the progression appears correct (all four lines are approximately parallel), the order parameter appeared to begin at the incorrect magnitude.

The fix corrected the computation of the nematic order parameter by adjusting the normalisation. In earlier versions, the identity term was subtracted as 3<u>I</u> (where <u>I</u> is the identity matrix) effectively removing the trace only once. Without this correction, the calculated order parameter was artificially inflated, giving misleadingly high values for disordered configurations. Instead L^2<u>I</u> was subtracted to remove the trace once per lattice site. This solved the problem in all three optimised scripts:

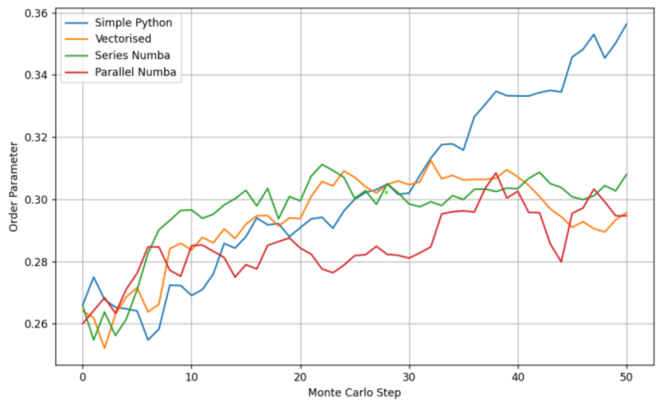


Fig. 4: Systematic error removed from optimised scripts.

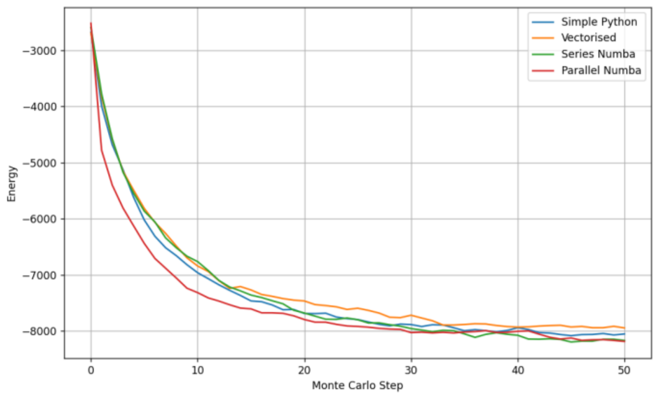


Fig. 5: Fixed scripts show no difference in behaviour.

4. Cython OpenMP: ‵ ‵_cy.py

Cython compilation allowed for true parallel multithreading. Many versions of this code were written and most did not compile. An initial script which did not take advantage of OpenMP saw minor performance gains over multithreaded Numba. The main advantage was found over small N, L, where JIT compilation overhead dominated total runtime.

The Cython code runs on the same chequerboard principle the Numba code, but using different syntax in the kernel. Development of this code comprised mainly trial-and-error and GPT was used to assist debugging in this case.

Once OpenMP was successfully implemented, performance increases over multithreaded Numba were substantial at high N, L. Using the updated common test parameters, OpenMP saw a total Runtime of 23.207138 s on a single thread reduce to **13.461785 s** on 4. This figure did not change running on all 12 available threads. With the same parameters, Numba on 4 threads achieved ~20 s, vectorised python achieved a time of 1079.514523 s and the raw python took > 3000 s.

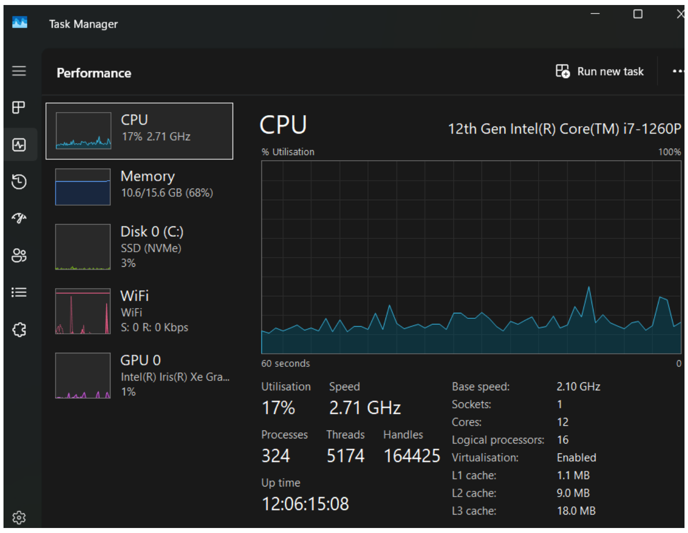
5. MPI: : ‵ ‵_mpi.py

The MPI method made use of a single-dimensional domain decomposition by columns. Each rank owns a contiguous block of columns to work on independently plus two ghost columns to exchange information between ranks. This method was used in order to preserve the physics as it guarantees no two nearest neighbours are updated simultaneously. Similar to previous chequerboard update methods, all red sites are updated using neighbours, then ghost columns are exchanged, then all black sites are updated, then ghosts are exchanged again.

In this hybrid-mode parallelised code, MPI is used to distribute columns (each MPI rank simulates a block of the lattice independently), OpenMP threads update individual lattice sites within one MPI rank, and NumPy is used to vectorise anything which cannot be done in parallel, such as I/O and RNG.

The Cython kernel is lightly modified from previous. They are logically identical.

This code was unsuccessful at speeding up the simulation with the higher parameters; the MPI ranks utilised all the system's available RAM, meaning the CPU was bottlenecked at about 10-20% utilisation.


Task Manager

Performance

Run new task

CPU 17% 2.71 GHz

Memory 10.6/15.6 GB (68%)

Disk 0 (C:) SSD (NVMe) 3%

WiFi WiFi S: 0 R: 0 Kbps

GPU 0 Intel(R) Iris(R) Xe Gra... 1%

CPU 12th Gen Intel(R) Core(TM) i7-1260P

% Utilisation 100%

60 seconds 0

Utilisation 17% Speed 2.71 GHz

Processes 324 Threads 5174 Handles 164425

Up time 12:06:15:08

Base speed: 2.10 GHz
Sockets: 1
Cores: 12
Logical processors: 16
Virtualisation: Enabled
L1 cache: 1.1 MB
L2 cache: 9.0 MB
L3 cache: 18.0 MB

Fig. 6: Task manager, RAM is the limiting factor.

With the lower parameters, and parameters up to circa 70, 100, this RAM overload was nor present and the script ran roughly as quickly as the NumPy vectorised script. In other words, in low parameters, the overheads dominate runtime, and with high parameters, the system has insufficient RAM to handle the MPI infrastructure. One might expect substantial speed increases on a high-performance PC.

Using the code without MPI ran approximately as fast as Numba.

The results of this code are unsatisfactory (it's the taking part that counts).

7. Simulation Results

The simulation behaviour went unchanged throughout each version. This is because care was taken to ensure parallel processing did not interfere with the physics of the simulation.

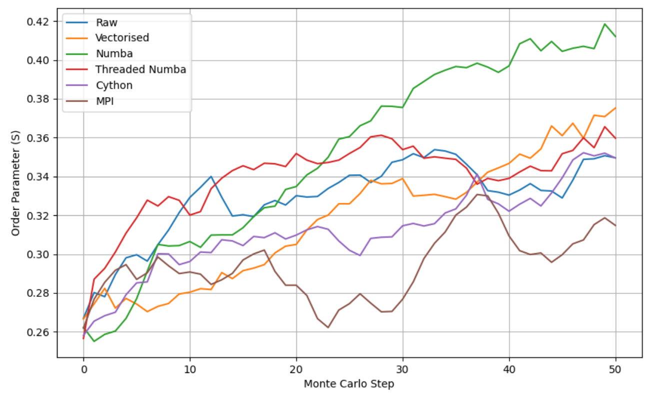


Fig 7: Subsequent versions did not shown unusual evolution of order parameter. This test would have been more comprehensive had these scripts been run with a randomisation seed.

8. Time Scalability

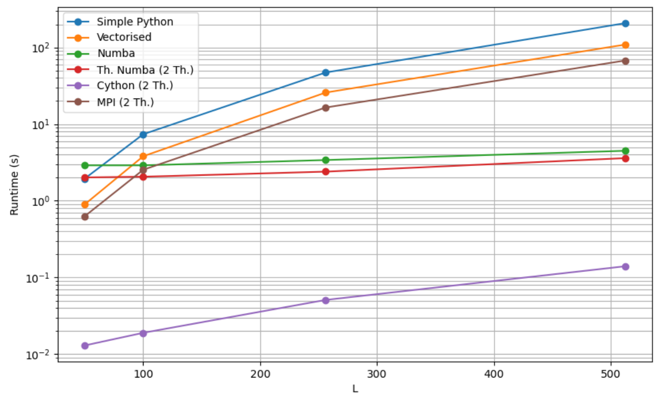


Fig. 8: Runtime versus L, lattice dimension, O(L^2) , for each script.

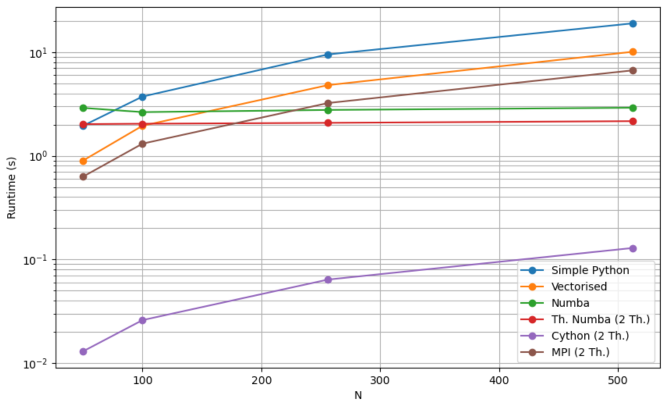


Fig. 9: Runtime versus N, MC steps, O(N) , for each script.

The advantages of Numba compilation become more apparent in higher runtimes; a relative speed up of 21× is a fourfold increase over the relative gain of 5× in the smaller parameters (*Tables 1,2*). The overheads caused by JIT compilation become negligible at high runtimes and threaded From *Fig. 8*, it appears Numba will outperform MPI and Vectorisation at around L=100, and from *Fig. 9*, the overheads become negligible at circa N=230. Numba may match/outperform Cython at larger scales than shown.

MPI is a complete, abject failure on this system, showing virtually no performance gain regardless of scale.

With common test parameters of 50, 50, bottlenecks appear once the overheads (eg. compilation, WSL thread overhead, displayed in brackets) dominate runtime:

Filename –".py"	Runtime / s	Rel. Gain
LebwohlLasher	**2.999688**	0
' '_vectorised	**0.989825**	3×
' '_numba	**0.609752 (+2)**	5× (~0)
' '_numba_p	**0.024596 (+2)**	121× (~0)
' '_cy	**0.038707**	77×
' '_mpi	**~0.7**	5×

Table 1: 50, 50, 0.5 0.

With common test parameters of 512, 512, these bottlenecks disappear as the overheads do not scale with the same time complexity:

Filename –".py"	Runtime / s	Rel. Gain
LebwohlLasher	3010.9	0
' '_vectorised	1079.5	3×
' '_numba	~ 140	21×
' '_numba_p	20.23019	150×
' '_cy	**13.461785**	225×
' '_mpi	**~ 2000**	~0

Table 2: 512, 512, 0.5, 0:

9. Version Control

GitHub version control was widely and comprehensively used throughout the development of this project.