**The Lebwohl-Lasher Model of Liquid Crystals**

1 Code modifications/NumPy vectorization

First of all, I tried to rewrite several functions in LebwohlLasher.py in various ways to run them with numpy.1) In one\_energy, I tried using numpy array and using np.sum, but it didn't work very well, and I speculate that because there are only 4 directions, the time and memory allocation needed to run numpy is more than the direct run itself.2) in all\_energy, the use of np.roll and vectorisation is very effective, the time required for all\_energy is reduced from 6.622 to 0.149 with the parameter: 500 50 0.5 0. However, in the subsequent process, np.roll can not be compiled into Cython, and so it is only used in Numba. 3) In get\_order, numpy is also very effective, with the parameter: 500 50 0.5 0, the time needed for get\_order is reduced from 5.983 to 0.150. 4) In MC\_step, in the original file, the option of randomly selecting the coordinates of the lattice is selected, which consumes a lot of time, especially in the cache, where every time you randomly select a coordinate, you need to fetch the cache once. So I chose to calculate the coordinates in order. Unfortunately, following the order doesn't make a lot of difference. With the parameter: 500 50 0.5 0, it only changed from 18.525 to 15.549.

1. Numba.

Numba is implemented by marking @jit(nopython=True) in front of important functions. but in all\_energy, it is not possible to automate np.roll, so I wrote a function roll. And added parallel=True to get\_order and MC\_step. It is worth noting that , the speedup is not noticeable in small scale tests, but it is noticeable in large scale runs.

python Numba-LebwohlLasher.py 50 50 0.5 0

Numba-LebwohlLasher.py: Size: 50, Steps: 50, T\*: 0.500: Order: 0.369, Time: 1.219966 s

python Numba-LebwohlLasher.py 500 50 0.5 0

Numba-LebwohlLasher.py: Size: 50, Steps: 500, T\*: 0.500: Order: 0.730, Time: 1.472361 s

1. Cython.

In Cython, I define the types of functions and variables via cdef while replacing numpy's functions with libc.math, but I have to discard all\_energy and get\_order, which were previously accelerated via array in numpy. re-run it with the same idea and code from the very beginning.

python run\_Cython\_LebwohlLasher.py 50 50 0.5 0

run\_Cython\_LebwohlLasher.py: Size: 50, Steps: 50, T\*: 0.500: Order: 0.332, Time: 0.068497 s

python run\_Cython\_LebwohlLasher.py 500 50 0.5 0

run\_Cython\_LebwohlLasher.py: Size: 50, Steps: 500, T\*: 0.500: Order: 0.539, Time: 0.649713 s

OpenMP(Cython)

When I called OpenMP with one\_energy, it didn't speed up.

with nogil:

for ix in prange(nmax, schedule='static', num\_threads=num\_threads):

en += 0.5 \* (1.0 - 3.0 \* cos(central\_value - arr[ixp, iy])\*\*2)

en += 0.5 \* (1.0 - 3.0 \* cos(central\_value - arr[ixm, iy])\*\*2)

en += 0.5 \* (1.0 - 3.0 \* cos(central\_value - arr[ix, iyp])\*\*2)

en += 0.5 \* (1.0 - 3.0 \* cos(central\_value - arr[ix, iym])\*\*2)

As an example, when I use only Cython in one\_energy，

python run\_OpenMP\_Cython\_LebwohlLasher.py **500** 50 0.5 0 4

run\_OpenMP\_Cython\_LebwohlLasher.py: Size: 50, Steps: 500, T\*: 0.500: Order: 0.187, Time: 0.962174 s

But when I call OpenMP after python run\_OpenMP\_Cython\_LebwohlLasher.py 50 50 0.5 0 4, the result is about 15s.

Similarly, when I don't use OpenMP in all\_energy

python run\_OpenMP\_Cython\_LebwohlLasher.py 500 50 0.5 0 4

run\_OpenMP\_Cython\_LebwohlLasher.py: Size: 50, Steps: 500, T\*: 0.500: Order: 0.187, Time: 0.962174 s

When I use OpenMP in all\_energy，

python run\_OpenMP\_Cython\_LebwohlLasher.py 500 50 0.5 0 **4**

run\_OpenMP\_Cython\_LebwohlLasher.py: Size: 50, Steps: 500, T\*: 0.500: Order: 0.196, Time: 10.505782 s

Instead of increasing the speed it decreases.

Interestingly, when I changed num\_threads, from 4 to 1, the speed increased instead.

python run\_OpenMP\_Cython\_LebwohlLasher.py 500 50 0.5 0 **1**

run\_OpenMP\_Cython\_LebwohlLasher.py: Size: 50, Steps: 500, T\*: 0.500: Order: 0.539, Time: 0.729980 s

I presume that instead of making it faster, multithreading in small functions would have additional overheads such as thread synchronisation. And in this code, one\_energy requires python, so it can't be set with nogil:, and these interactions have extra costs. And one\_energy itself only has four values, in practice, the communication cost is higher. So I gave up on the OpenMP acceleration.

However, what puzzles me is that when I use OpenMP in MC\_step, the speed is also reduced.

python run\_OpenMP\_Cython\_LebwohlLasher.py 500 50 0.5 0 4

run\_OpenMP\_Cython\_LebwohlLasher.py: Size: 50, Steps: 500, T\*: 0.500: Order: 0.152, Time: 18.754455 s

No OpenMP：

python run\_OpenMP\_Cython\_LebwohlLasher.py 500 50 0.5 0 4

run\_OpenMP\_Cython\_LebwohlLasher.py: Size: 50, Steps: 500, T\*: 0.500: Order: 0.199, Time: 1.023467 s

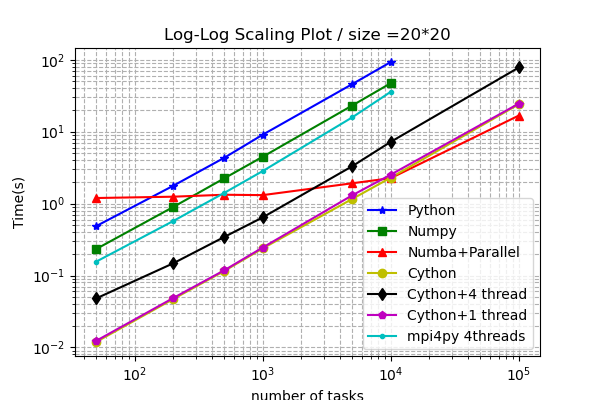
python run\_OpenMP\_Cython\_LebwohlLasher.py 500 50 0.5 0 1

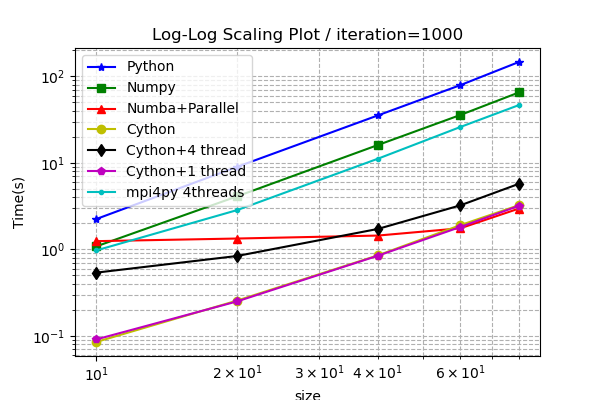
run\_OpenMP\_Cython\_LebwohlLasher.py: Size: 50, Steps: 500, T\*: 0.500: Order: 0.539, Time: 0.660428 s

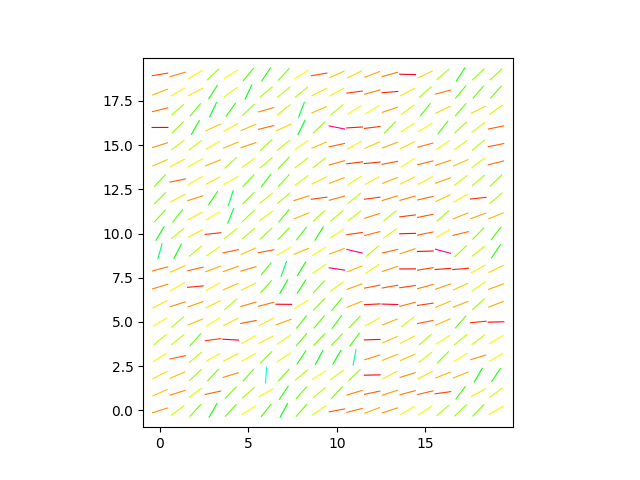
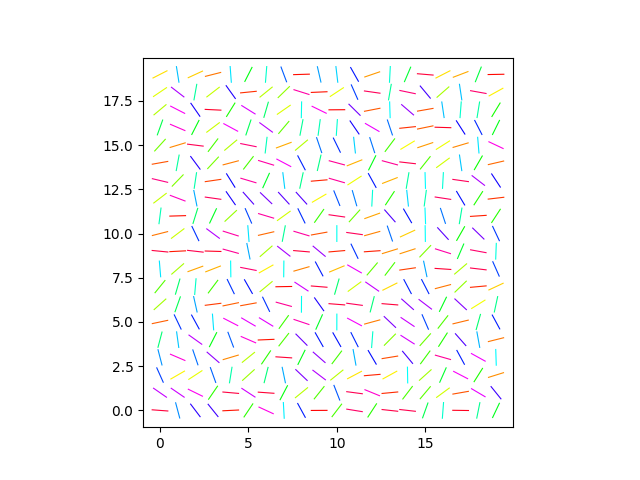
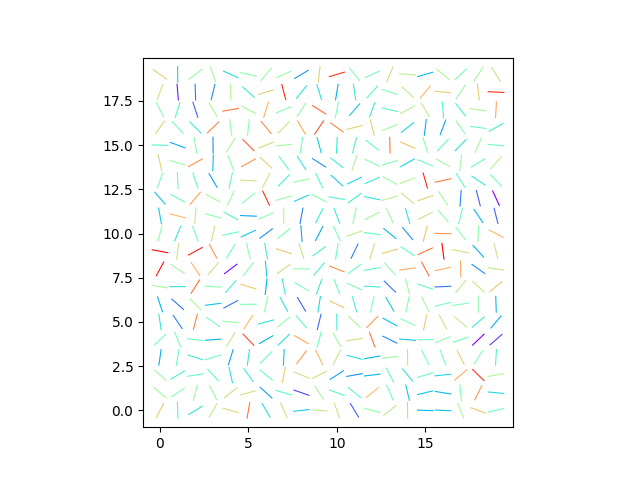
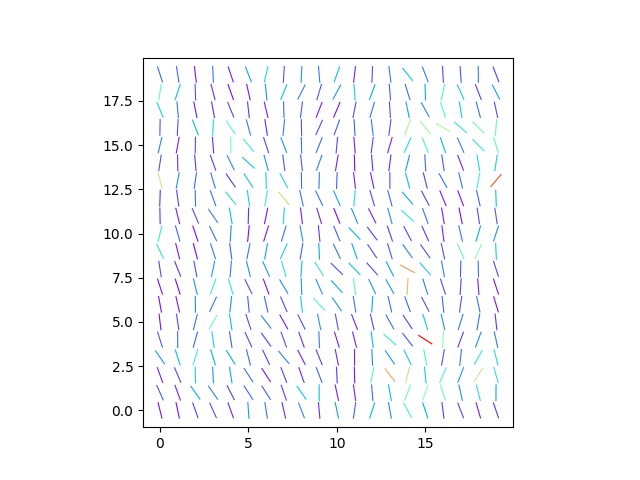
1. mpi4py
2. conclusion

我最喜欢numba。方便操作，效率很高。关于parallel，要看互相有没有沟通。并没有看到明显的加速，相对python有很大进步，毕竟这个是基础。

speed comparisons and scaling plots







Github: