Introduction to High Performance Computing
Active Matter Time Optimization
Expected Grade C

https://github.com/druskus20/ithpc-project

Pedro Burgos Gonzalo
pedrobg@kth.se

15-03-2024

Introduction

The study of active matter systems, such as bird flocks, or bacterial colonies, is of significant interest due to their complex collective behaviors and potential applications in various scientific fields such as ecology or medicine. These systems consist of numerous individual agents interacting with each other and their environment, exhibiting emergent behaviors that cannot be understood by studying each agent in isolation. Instead, we can turn to computational models to simulate and analyze the dynamics of these systems.

In this case I will focus on the simulation of bird flocks, through the use of the Viscek model (1995). This model is based on the assumption that each bird in a flock adjusts its velocity to match the average velocity of its neighbors. The Viscek model captures the essential features of bird flocks.

The starting point will be a baseline simulation code provided by Philip Mocz. The goal of this project is to implement and benchmark various optimization techniques it. Specifically I aim to enhance the *time* performance of the algorithm.

Moreover, understanding the dynamics of bird flocks has implications beyond the realm of biology. By studying flocking behavior, we can gain insights into swarm intelligence, distributed computing, and decentralized control systems. These concepts are relevant to a wide range of disciplines, from artificial intelligence to urban planning. Therefore, by improving the efficiency of bird flock simulations, we potentially advance our understanding of biological systems but also contribute to the development of innovative solutions.

Experimental Setup

All the tests shown in this reports were performed on a modern laptop with the following specifications:

- Ubuntu Server 22.04.3 LTS
- Linux 5.15.0-92-generic x86_64
- AMD Ryzen 7 PRO 5850U, 8c 16t
- 16 GB Ram
- 512GB SSD

The code lies both in a local git repository and in a remote one, hosted on GitHub. The code was developed and tested using Python 3.10.12.

The following versions of the main libraries were used:

- numpy==1.24.2
- Cython==3.0.8
- line-profiler==4.1.2

For each graph shown, the results are averaged out of 10 executions. The standard deviation is also calculated in an aim to showcase any irregularity.

Methodology

To commence, I initiated the project by refining the baseline simulation code to enable efficient profiling and benchmarking. I also removed non essential features like the real time animation - which I implemented in a separate code - and added at a later stage.

I utilized CProfile and Line Profiler, to analyze the baseline simulation code. These tools enabled a granular examination of the codebase, pinpointing specific functions and lines of code contributing significantly to execution time. This allowed me to confirm that the bottleneck of the code happens in the inner loop of each step.

```
# find mean angle of neighbors within R
mean_theta = theta
for b in range(N):
    neighbors = (x-x[b])**2+(y-y[b])**2 < R**2
    sx = np.sum(np.cos(theta[neighbors]))
    sy = np.sum(np.sin(theta[neighbors]))
    mean_theta[b] = np.arctan2(sy, sx)
```

Figure 1: Inner loop of the simulation

Line profiler clearly shows that the bulk of time I spent here, and thus I decided that this would be the entry point of my optimization efforts.

```
83       300    1060328.0   3534.4      0.1      x += vx*dt
84       300     854504.0   2848.3      0.1      y += vy*dt
85
86                                               # apply periodic BCs
87       300    1337181.0   4457.3      0.2      x = x % L
88       300    1035267.0   3450.9      0.1      y = y % L
89
90                                               # find mean angle of neighbors within R
91       300     397192.0   1324.0      0.1      mean_theta: np.ndarray = theta
92     30300   38698662.0   1277.2      4.9      for b in range(N):
93     30000  214300914.0   7143.4     27.0          neighbors: np.ndarray = (x-x[b])**2+(y-y[b])**2 < R**2
94     30000  225718056.0   7523.9     28.4          sx: float = np.sum(np.cos(theta[neighbors]))
95     30000  212039114.0   7068.0     26.7          sy: float = np.sum(np.sin(theta[neighbors]))
96     30000   87646065.0   2921.5     11.0          mean_theta[b] = np.arctan2(sy, sx)
97
98                                               # add random perturbations
99       300    2306014.0   7686.7      0.3      theta = mean_theta + eta*(np.random.rand(N,1)-0.5)
100
101                                              # update velocities
102      300    1239534.0   4131.8      0.2      vx = v0 * np.cos(theta)
```

Figure 2: Line profiler results for the inner loop

Next, I refactored the code to allow for easy modification and testing, as well as increased clarity. I tried different alternatives, such as using a class to hold global mutable state, or using matplotlib's `FuncAnimation` to animate the simulation at each step. However I quickly discarded all of these ideas in favor of simplification, as I quickly realized the performance impact of either of them was too high. I also added type annotations to all the variables and parameters, and avoided pass-by-copy variables when possible.

Furthermore, I converted most of the operations to use Numpy's vectorized functions, as they are potentially faster than the equivalent for loops.

The end result was a simple, clear, and easy to understand code, that is easy to modify and test. I also designed a Jupyter notebook to visualize and compare benchmarking results. This platform facilitated the iterative analysis and comparison of different code versions and optimization techniques.

```python
def measure_simulation(name, simulation_fn, sim_parameters, N_SIM=10, animate=False, desc=""):

    step_times_per_run = []  # Store average step times for each step
    total_times_per_run = []  # Store total times for each simulation

    for _ in range(N_SIM):
        # Simulate
        t_start = time_ns()
        positions, velocities, step_times = simulation_fn(sim_parameters)
        total_time = time_ns() - t_start

        if animate:
            flock_simulation.animate(name, sim_parameters, positions, velocities)

        total_times_per_run.append(total_time)
        step_times_per_run.append(step_times)

    # Calculate average step times and standard deviation across all runs
    avg_step_times = np.mean(np.array(step_times_per_run), axis=0) / 1e9  # Convert to seconds
    std_step_times = np.std(np.array(step_times_per_run), axis=0) / 1e9  # Convert to seconds

    # Calculate average total time and standard deviation across all runs
    avg_total_time = np.mean(total_times_per_run) / 1e9  # Convert to seconds
    std_dev_total_time = np.std(total_times_per_run) / 1e9  # Convert to seconds

    TIMES[name] = {
        'desc': desc,
        'avg_step_times': avg_step_times,
        'std_step_times': std_step_times,
        'avg_total_time': avg_total_time,
        'std_dev_total_time': std_dev_total_time
    }
```

Figure 3: Function to measure the simulation time

For testing, I utilized a parameter grid method, systematically exploring various combinations to assess their impact on performance. During this process, I identified two relevant parameters: Nt (number of steps) and N (number of birds). Altering these parameters could potentially showcase problem specific optimizations. I decided on a regular distribution for each of them, based on my previous experience running the code on my hardware.

I also calculated the average and standard deviation of 10 executions per result. I measured both the total time it took to execute the simulation, and the time per step.

```python
import itertools

# Define simulation parameters
v0 = 1.0          # velocity
eta = 0.5         # random fluctuation in angle (in radians)
R = 1             # interaction radius
dt = 0.2          # time step
L = 10            # size of box
Nt = [30, 60, 120, 240]  # number of time steps
N = [100, 200, 400, 800]  # number of birds

# Generate all possible combinations of simulation parameters and assign names
SIM_PARAMETERS = {}
for Nt_val, N_val in itertools.product(Nt, N):
    name = f"Nt_{Nt_val}_N_{N_val}"
    SIM_PARAMETERS[name] = (v0, eta, L, R, dt, Nt_val, N_val)
```

Figure 4: Parameter grid

The first optimization method that I tried was parallelization, I used the multiprocessing module to parallelize the inner loop of the simulation with a `ThreadPoolExecutor`. In this way, every step is executed sequentially, as it depends on the previous one, but the birds calculate the next position in parallel.

```python
def bird(b):
    neighbors = (x - x[b]) ** 2 + (y - y[b]) ** 2 < R**2
    sx: float = np.sum(np.cos(theta[neighbors]))
    sy: float = np.sum(np.sin(theta[neighbors]))
    return np.arctan2(sy, sx)

with ThreadPoolExecutor(max_workers=8) as executor:
    mean_theta = np.array(list(executor.map(bird, range(N))))
```

Figure 5: Parallelization with ThreadPoolExecutor

The second optimization method that I tried was partial compilation to C. Utilizing Cython, I was able to transpile different parts of the code to C. I attempted to compile the entire program at first, as well as the entire step function, however, it required a significant effort to gain performance benefits this way, so I decided to focus my efforts even further. In particular, I offloaded the mean theta calculation, to take advantage of libc's arithmetic functions.

I also modified the function to write directly into an array instead of returning a new one, and I used the `cdef` keyword to declare the types of the variables. Finally I used the `nogil` keyword to release the GIL, and the `boundscheck` and `wraparound` keywords to disable bounds checking and negative indexing.

```
@cython.boundscheck(False)
@cython.wraparound(False)
def calculate_mean_theta(np.ndarray[np.float64_t, ndim=2] theta,
                         np.ndarray[np.float64_t, ndim=2] x,
                         np.ndarray[np.float64_t, ndim=2] y,
                         double R,
                         np.ndarray[np.float64_t, ndim=2] mean_theta
                        ) nogil:
    cdef Py_ssize_t N = theta.shape[0]
    cdef Py_ssize_t b, i
    cdef double sx, sy

    for b in range(N):
        sx = 0
        sy = 0
        for i in range(N):
            if (x[b, 0] - x[i, 0]) ** 2 + (y[b, 0] - y[i, 0]) ** 2 < R ** 2:
                sx += cos(theta[i, 0])
                sy += sin(theta[i, 0])

        mean_theta[b, 0] = atan2(sy, sx)

    return mean_theta
```

Figure 6: Cython compilation

Results

First of all, I was able to conclude that altering the parameter space did not have a significant impact on performance. After increasing either N or Nt the increase in execution time obtained was regular and uniform. Obviously, a greater value of N, implies higher deviation since bigger arrays are employed, but the average time was consistent.

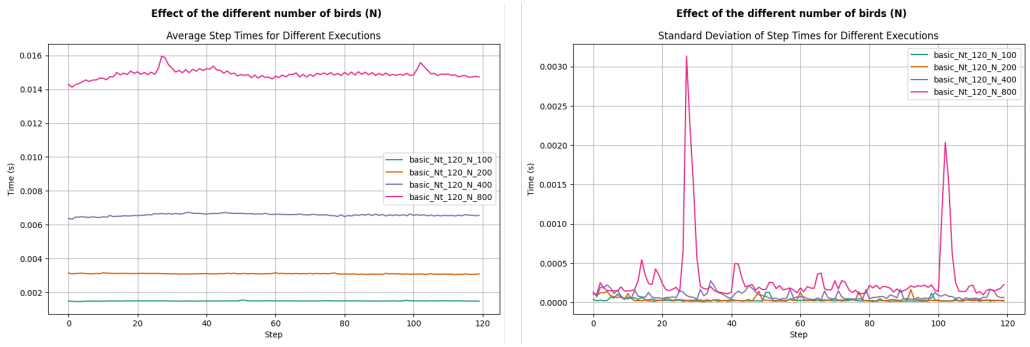

Figure 7: Per-step time average and standard deviation when changing N

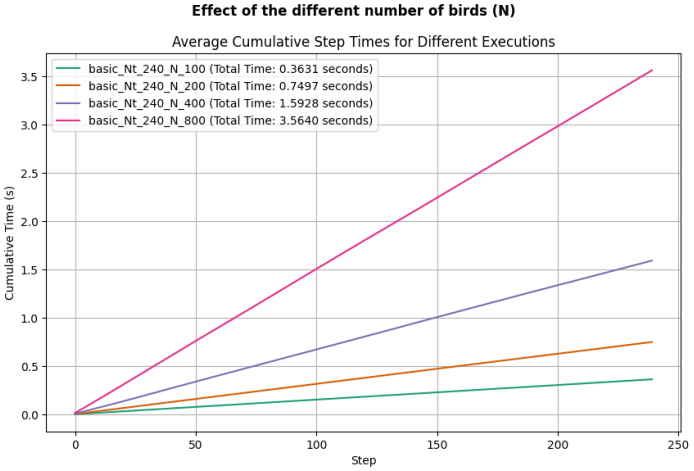

Figure 8: Cumulative average time per step when changing N

Both parameters, N and Nt were tested independently, and the results were consistent in the total time taken. Increasing either of them resulted in a proportional and uniform increase in the total time taken to execute the simulation.

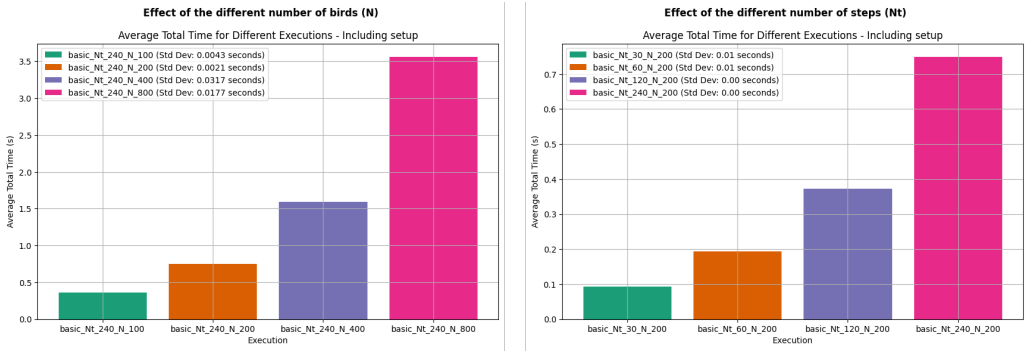

Figure 9: Average total time when changing N and Nt

However, when comparing the different versions of the algorithms, the results were clear. The Cython version performs much better than the original, by a factor of 9.8x approximately.

To understand the results, it is important to understand the terminology employed in the graphs. There are 4 names assigned to each of the versions of the algorithms:

- original: The baseline code provided by Philip Mocz, with the animation removed.
- basic: Basic refactor over the original, utilizing Numpy and restructuring the code.
- parallel: The parallelized version of the basic code using `ThreadPoolExecutor`.

- cython: The Cython compiled version of the basic code, with the `mean_theta` calculation offloaded to C.

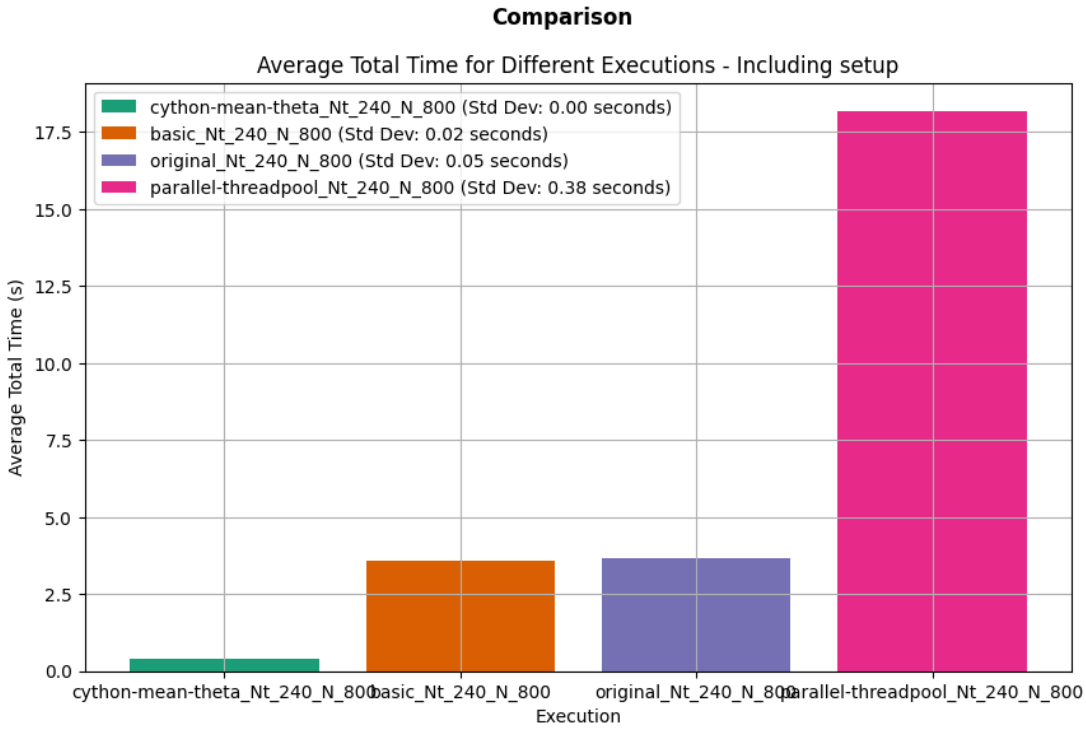

Figure 10: Average total time for the different optimizations

The parallel version is counterproductive, as the overhead of creating and managing threads is too high, and this is reflected in the standard deviation and the total time taken.

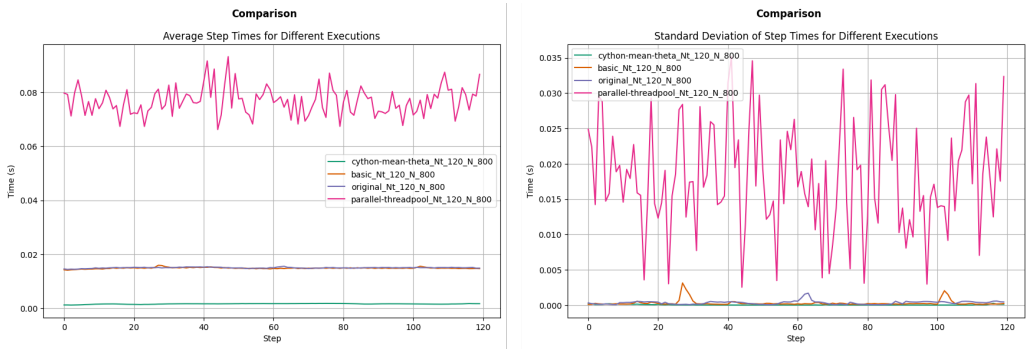

Figure 11: Time per step and standard deviation for the different optimizations

In Figure 11, we can see the high variability of the parallel version, compared to the other versions. It is worth noting that utilizing a higher number of workers, or switching to a `ProcessPoolExecutor` did not yield any significant improvements. It is possible that in a different hardware, diverse results could be obtained.

Note that I measured both the total execution time, as well as the cumulative time per step, in order to identify if a bottleneck was present on the actual algorithm, or the boilerplate code that facilitates the profiling and benchmarking. However, I was able to verify that this, did not have a significant impact on the performance.

Discussion and Conclusion

Evaluating the average time and the standard deviation across several executions of the simulation contributes to reduce the impact of outliers.

We have concluded that, in this particular case, the original code was quite efficient, as it was short and simple. The parallelization method did not yield any significant performance improvements, since the overhead of creating and managing threads was too high.

The Cython compilation method did greatly improve the performance of the inner loop by a **factor of ~9.8x** approximately, and we established that further C compilation is generally not worth the effort in maintainability and complexity. Small improvements were achieved by disabling the GIL and adding type annotations everywhere, however, the biggest improvement was the use of libc's arithmetic functions.

The main alternative technique left to explore is the use of GPU acceleration, through Pytorch, Cupy or another library of a similar nature. This would allow us to take advantage of the massive parallelism of the GPU to calculate the next position of the birds and possibly obtain a significant performance improvement. In my case, I decided to leave it out of the scope given that I currently do not have access to a stable internet connection due to traveling, or an Nvidia GPU.

The code and the notebook are available in a public repository at Github