Advanced Computational Physics: Boids

https://github.com/V-Hill/advanced-computational-physics

V. Hill

University of Bristol

1st March 2021

Abstract

In this project I have implemented the Boids flocking behaviour simulation. Using the Python programming language, I explore techniques to speed up the run-time of this simulation. I investigate both shared and distributed memory parallel processing architectures using the MPI and OpenMP frameworks. I make use of the BlueCrystal3 high performance computing system available at Bristol to run code on up to 16 cores. I explore the effects of different data structures on code performance, and employ Cython to generate precompiled C modules. Finally, I use a divide and conquer triangulation algorithm to reduce the computational complexity of the boids programme.

1 Boids

In 1986, Craig Reynolds developed an artificial life programme which simulates the flocking behaviour of birds. Reynolds employed the use of the word 'boid', which represents a shortened version of 'bird-oid object'. In his seminal paper [1] 'Flocks, Herds, and Schools: A Distributed Behavioral Model' Reynolds describes a set of simple rules which specify the behaviour of each individual boid. From the collective interaction of these boids, following only simple rules, the complex behaviour of flocking arises.

Flocking is an example of the phenomena of emergence. In physics this describes systems observed to have properties which its individual parts do not have on their own; properties which emerge only when the parts interact as a whole [1].

[1]The flock is greater than the sum of its birds

1.1 Boid : Rules

The flocking behaviour of the Boids simulation results from three simple rules - separation, cohesion and alignment. Each boid acts independently and changes its speed and direction in response to its neighbouring boids.

The *separation* rule means that boids must steer to avoid crowding local neighbouring boids. If a neighbouring boid comes too close, the direction of travel is altered to avoid collisions.

The *cohesion* rule states that boids must steer to move toward the average position of their local neighbourhood of boids. This encourages flocks to form.

The *alignment* rule states that boids must steer towards the average heading of their boid neighbourhood. This encourages boids to travel together.

As an additional rule, each boid can also have a limited field of vision, only taking into account boids within a certain angle either side of its travel direction.

2 Nearest neighbour search - Linear search

As the behaviour of each boid is based solely on observations of its nearest neighbours, the first challenge is to determine the set of nearest neighbours for each boid (the boids 'neighbourhood').

Nearest neighbour search (NNS) is an optimisation problem which forms the basic building block of numerous computational geometry algorithms. Although conceived of much earlier [2], the problem was formally defined by Donald Knuth in 1973 as the following:

$$\text{Given a set } S \text{ of } n \text{ points in a space } M \text{ and a query point } q \in M \text{ find the closest point in } S \text{ to } q \tag{1}$$

In the case of the Boids programme, we want to find a set of multiple nearest neighbours. Numerous solutions to the NNS problem have been proposed, with algorithms varying in time and space complexity. I start this report by investigating the simplest solution, to obtain a comparative baseline for further more advanced methods.

The simplest solution to the NNS problem is to iterate through all of the n many points, computing the Euclidean distance from the query point x_q to each other point x_i. The point x_i is a neighbour of x_q if the following is satisfied:

$$\text{Neighbour if:} \quad \sqrt{(x_q - x_i)^2 + (y_q - y_i)^2} < D_{thres} \tag{2}$$

To construct a set of nearest neighbours one can simply keep a list of the points with a distance less than some defined threshold D_{thres}. This returns the set of points within a fixed-radius of the query point. To find the neighbours of every point, this process is repeated for each point. This function was implemented in Python, using a nested for-loop and Python list data structure. Running on a single core of an Intel Xeon E5-2670 processor within the BlueCrystal3 HPC system, a run-time of 79.7±0.6s for computing all neighbours of 10,000 boids was measured.

2.1 Python optimisations

In computing, a processing unit is limited in speed by the number of operations it can perform in a given unit of time. Floating point operations per second (FLOPS) measure a computer's performance in terms of instructions per second, with modern systems performing billions of FLOPS. For reference, the Nvidia K20 GPUs in BlueCrystal3 have a peak double precision floating-point performance of 1.17 teraFLOPS. One of the simplest forms of programme optimisation is to reduce the number of operations performed during the programme. Less calculations means less execution time.

The neighbour test in equation (2) can be modified to reduce the number of operations by using:

$$\text{Neighbour if:} \quad (x_q - x_i)^2 + (y_q - y_i)^2 < D_{thres}^2 \tag{3}$$

This yield the same result, but removes the need to compute the computationally

[2]The idea of nearest neighbour search appears as early as the 11th century. It is described in the context of visual object recognition in the medieval text 'Book of Optics' by acclaimed Arab scholar Alhazen [2].

expensive square root n^2 times during the programme. This can be further optimised by using the Manhattan distance metric, implemented as follows:

$$\begin{aligned}\text{Neighbour if:}\quad & -D_{thres} < (x_q - x_i) < D_{thres} \\ \text{and}\quad & -D_{thres} < (y_q - y_i) < D_{thres}\end{aligned} \tag{4}$$

This removes having to perform the two multiplications present in equation 3. However, rather than returning the set of fixed-radius neighbours, this approach finds neighbours within a square centred on the query point. A comparison of the speed-up gained with these optimisations is shown in table 1.

Table 1: Table of the run-time speed-up gains using different distance measures: [a]equation 2, [b]equation 3, [c]equation 4
Computing all neighbours of 'number of points' many boids.

	Speed-up relative to Euclidean distance		
Number of points	Euclidean distance[a]	Euclidean squared[b]	Manhattan distance[c]
100	1.00	1.12 ± 0.01	2.59 ± 0.01
1000	1.00	1.13 ± 0.02	2.52 ± 0.05
10000	1.00	1.14 ± 0.01	2.51 ± 0.02

With these modifications to the neighbour criterion a roughly 2.5× speed-up was observed (table 1).

2.2 MPI parallelism

Parallel processing allows a programme to be broken up into separate tasks and shared across multiple processors. The Message Passing Interface (MPI) standard provides a standardised protocol for communication between these multiple processors. MPI is used in distributed memory architectures, where each processor has its own private memory. With MPI, a programme can be run across many multi-core processors on separate computing nodes. This enables a programme to be efficiently scaled and take advantage of modern multi-node supercomputers.

In this project I have used the MPI for Python (mpi4py) library to parallelise the linear search algorithm. The linear search method can be parallelised by dividing the query points into n equally sized groups, and then splitting these groups across n many processing cores. Results of the parallelised Manhattan distance algorithm are shown in figure 1. In the idealised case, with n many cores, this would provide a speed-up equal to n.

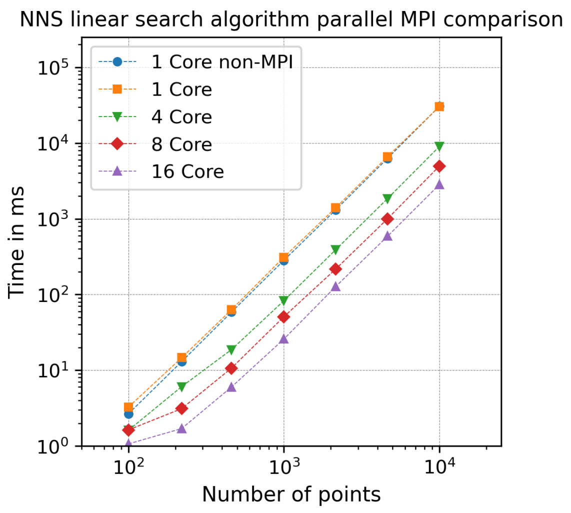

Figure 1: Timings for the linear search NNS algorithm using the Manhattan distance algorithm to find all boid neighbours. Using the OpenMPI library in Python, running on BlueCrystal3.

However, overhead due to amount of time processes spend communicating with each other limits the speed-up. This is evident when using a single core, where the MPI implementation took around 9% longer than the non-MPI version. With 100 points, the 16 core MPI version was only 3.1× faster than the single core version. Above 460 points, 16 cores was consistently around $10.5 \pm 0.8\times$ faster than a single core. Overall, with every factor of 2 increase in the number of cores, the programme was $1.7 \pm 0.2\times$ faster.

Compared with the baseline execution time for 10,000 boids of 79.7 ± 0.6s, the improvements so far have reduced the execution time by 28.2× to just 2.8 ± 0.3s. Next I seek to improve this further using Cython and shared-memory parallelism.

2.3 Cython and parallelism with OpenMP

2.3.1 Cython

Cython is an intermediate language between Python and C/C++ which adds C data types to Python code. Cython allows modified Python code to be written which gets compiled into C code. This compiled C code can then be called directly from a Python script. Variables can be defined as C-like types, something which is not available in the dynamic language of Python. As such, Cython can offer a significant speed-up with only minor modifications to a Python script. Cython also supports native parallelism, compatible with OpenMP, through the 'cython.parallel' module. Using shared-memory, OpenMP is used for parallelism within a multi-core node. In contrast, MPI allows for parallelism between nodes via its distributed memory approach.

In this section I implemented the nearest neighbour search in a object-orientated approach. This consisted of a Python class representing the boids, and a class function for finding the neighbours of all boids. The pure Python implementation of this performed worse than the initial baseline execution time for 10,000 boids, taking around 57% longer to execute at 125.0 ± 0.8s (as shown in table 2).

The linear search was then written in Cython. As much of the code was cythonised as possible, using all type defined variables. Numpy arrays were used to represent the boid positions and members of each boid neighbourhood. Additionally bounds-checking within these numpy arrays was disabled. Disabling bounds-checking removes unnecessary operations to check that each requested index is within the defined valid range of indexes for a given array. As this code relies heavily on array indexing, the removal of bounds-checking had a significant speed-up of $\approx 2\times$ depending on the setup. Combining these improvements, for 1000+ points, the execution time of the Cython code was $104 \pm 8\times$ faster than the pure Python code. This is remarkable speed-up, demonstrating clearly the performance gains available with pre-compiled C code. This Cython version is still only utilising a single core. Next, OpenMP enabled multi-processing was investigated.

2.3.2 Parallel Cython with OpenMP multi-processing

Given a single boid, we wish to check every other boid and test whether it is within a certain radius. The order in which we check each boids distance does not matter in this case. Thus when performing this loop the Python GIL (Global Interpreter Lock) can be disabled. The GIL serves the purpose of protecting Python objects by preventing multiple threads or processes from accessing the same memory simultaneously. By modifying the code such that each neighbour test only writes memory to a unique index I was able to disable the GIL.

With Cython, OpenMP multi-processing is easily added by using 'prange' (parallel range) inplace of the standard 'range' function. Using the 'nogil' argument to disable the GIL, the prange loop enables an OpenMP parallel for

Number of points	1 thread - Pure Python	1 threads - Cython	1 threads - Cython + OpenMP	2 threads - Cython + OpenMP	4 threads - Cython + OpenMP	8 threads - Cython + OpenMP	16 threads - Cython + OpenMP
100	12.96	1.01	0.99	0.54	0.29	0.16	0.08
1000	1212.19	12.36	28.95	14.60	7.39	3.79	1.92
10000	124972.74	1183.48	2591.72	1310.24	674.38	339.98	172.95

Table 2: Table of execution times (in ms) for the linear search algorithm implemented using pure Python, Cython and Cython+OpenMP, across a range of core counts.

loop to independently calculate each boid distance. OpenMP automatically starts a thread pool and distributes the workload evenly across processors. These threads are then distributed across the CPU cores and processed in parallel to save time.

2.3.3 Hyperthreading and Non-Uniform Memory Access

Modern CPUs are able to run multiple threads simultaneously on a single CPU core via Hyperthreading (for Intel CPUs) and Simultaneous Multi-Threading (for AMD CPUs). More threads means that more work can be done in parallel. In the case of the Xeon E5-2670 CPUs in Bristol's BlueCrystal 3 supercomputer, each of the 8 physical cores works like two 'logical cores' to handle different software threads, enabling a total of 16 concurrent threads per CPU.

Each BC3 node contains two Xeon CPUs, mounted in a dual-processor motherboard. In multiprocessing, both processors can access all of the system RAM, enabling the shared memory environment of OpenMP to be used. Additionally, each processor has local memory which it accesses directly. This memory is also part of the shared memory. However, memory access time depends on the memory location relative to a processor [3]. This leads to a Non-Uniform Memory Access (NUMA) enviroment, in which transferring data between threads on separate CPUs can incur a substantial delay. OpenMP does not provide explicit support for NUMA, and does not taking into account the relative access times of separate CPU registers when distributing threads [4]. This can lead to latency which limits the speed gains available in multi-CPU multiprocessing.

2.3.4 Results

Raw timings comparing pure Python to Cython and Cython+OpenMP are shown in table 2. It was found that, on a single thread, Cython with OpenMP performed worse than just Cython. At $2.3 \pm 1.1\times$ slower on average, this seems like a significant run-time penalty. However, the two thread performance of Cython with OpenMP was on par with the single thread Cython implementation. Furthermore, with every factor of 2 increase in the thread count, the run-time fell by $1.95 \pm 0.05\times$. As such, with 16 threads the Cython with OpenMP version was $14.6 \pm 1.3\times$ faster. For comparison, the MPI parallelised code saw only a $9.6 \pm 1.2\times$ drop in run-time with 16 cores. This shows that for the linear search algorithm, OpenMP parallelism scales more efficiently than parallel MPI code. The likely reason for this is that in OpenMP, threads share the same memory address space, so communications between threads are very efficient. In MPI model, each process has its own memory address space, and tasks exchange data by sending and receiving messages. This inherently adds computational overhead leading to the worse performance scaling of MPI compared to OpenMP[5].

2.4 Improving on linear search - computational complexity theory

The linear search method for nearest neighbour search could be efficiently scaled up further by adding additional cores; but we can do better. Expressed using big O notation, the linear search algorithm has time complexity $O(n^2)$ (even when parallelised), where n is the number of points. As seen in figure 1, each factor of 10 increase in the number of points increases the run time by ≈ 100 times.

For a large enough value of n an algorithm with less computational complexity, such as an $O(nlog(n))$ algorithm, will always outperform the linear search. The question then is, what is the most computationally efficient algorithm we could use for generating reasonable neighbourhoods of boids? In this report I explore the idea of point set triangulation as a solution to this.

3 Nearest neighbour search - Delaunay triangulation

Delaunay triangulation is a particular way of joining a set of points to make a triangular mesh. The technique was invented in 1934 [6] and has the following properties. For a set P of points in the plane, the Delaunay triangulation is the triangulation $DT(P)$ such that no point in P is inside the circumcircle of any triangle in $DT(P)$ (as illustrated in figure 2). As such, the Delaunay triangulation maximizes the minimum angle among all possible triangulations of that vertex set.

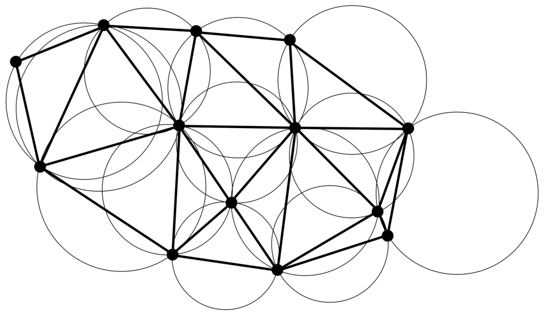

Figure 2: Every triangle of a Delaunay triangulation has an empty circumcircle [7].

3.1 Application to nearest neighbour search

The connection with NNS is that the nearest neighbour n to any point p is an edge np in the Delaunay triangulation. Hence, the nearest neighbour graph is a sub-graph of the Delaunay triangulation. In a 2D plane, each point has an average of six connected edges. The key point is that these edges can be used as a set of nearest neighbours. Hence, in finding the Delaunay triangulation, a set of nearest neighbours is found for all points (see figure 3).

3.2 Delaunay triangulation algorithm

Many algorithms for computing Delaunay triangulations exist, with varying computational complexity. I have chosen to implement a divide and conquer algorithm, as this can be parallelised very efficiently. Additionally, the divide and conquer approach has been shown to be the fastest class of Delaunay triangulation techniques [8].

The divide and conquer algorithm for Delaunay triangulation was originally developed in a 1985 paper by Guibas and Stolfi [9]. The basic technique for triangulating a set of 2D points can be summarised as follows [10]:

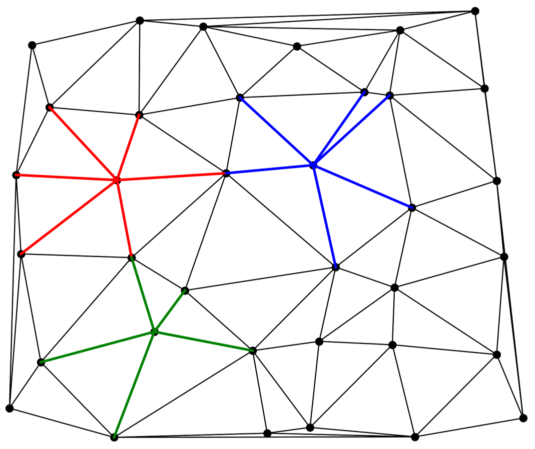

Figure 3: This figure shows the nearest neighbours of 3 points picked at random in the Delaunay triangulation of 36 points. The nearest neighbours of each point are given by the points connected edges.

1. The set of points is successively divided into halves, until we are left with subsets containing no more than three points.
2. These subsets are then directly triangulated. A line segment in the case of two points; a triangle in the case of three points.
3. The triangulated subsets are then recursively merged with their former other halves.
4. After all subsets are merged the triangulation is complete.

A visualisation of this process for 12 points is shown in figure 4.

The merge operation can be done efficiently in $O(n)$ time. The number of merge operations is proportional to $O(log(n))$. This gives a total running time of $O(n\,log(n))$, far superior to the linear search $O(n^2)$ time.

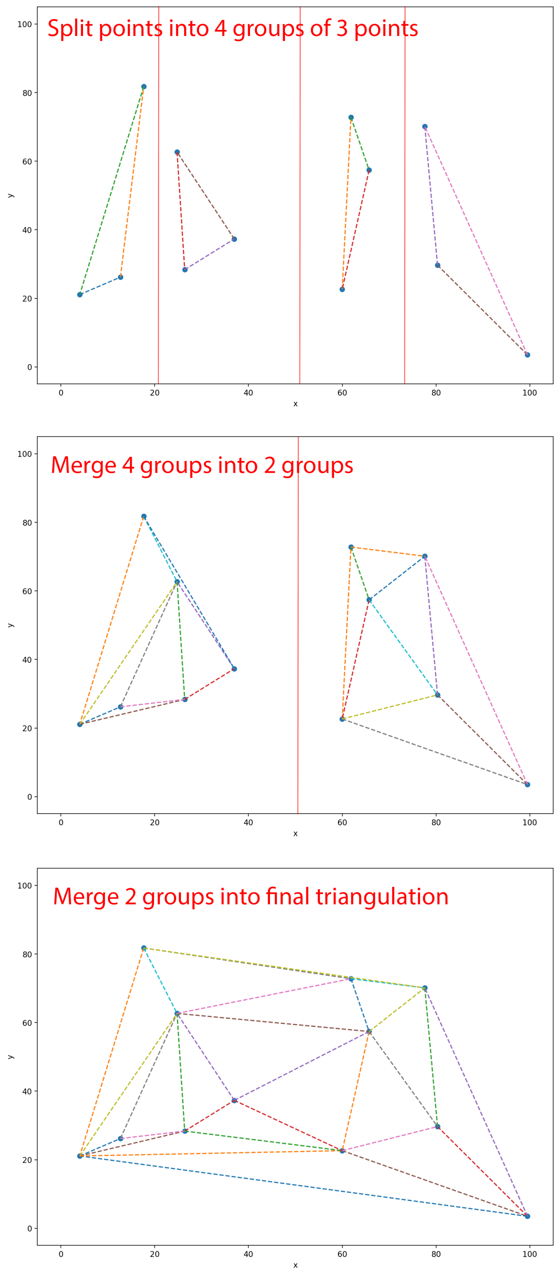

Figure 4: Plot showing the stages of the Guibas and Stolfi divide and conquer algorithm for Delaunay triangulation on a set of 12 randomly distributed points.

3.3 Delaunay triangulation for Boids

Delaunay triangulation (DT) has been used in swarm simulations, and is employed to effectively resolve collision optimisation problems [11]. I have not found any published research in which the divide and conquer algorithm has been parallelised for this task. As such, I present a novel method for simulating the Boids programme for the high performance computing environment.

3.4 Python implementation

Python contains a variety of data types. The choice of which representation to use affects both the performance and readability of the code. An edge in the triangulation is represented by 7 variables in the Guibas and Stolfi notation. I investigated storing these variables within a python list, a numerical representation in a numpy array and as a custom Python class. Results of the triangulation timings are shown in figure 5.

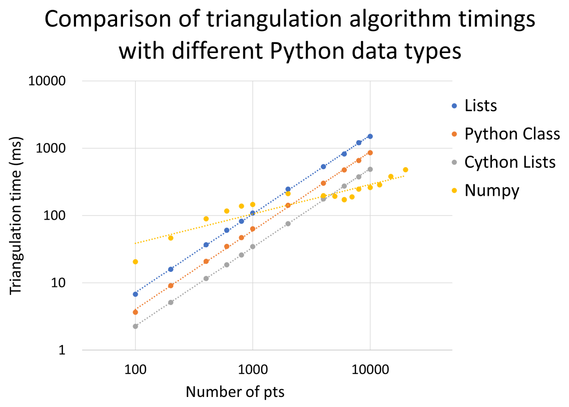

Figure 5: Comparison of Delaunay triangulation algorithm implementations using different python data types.

It was found that a numpy array provided the highest performance, lists the slowest, and a Python class in-between. In order to aid in the code development and maximise readability, I chose to go with an object-orientated class implementation.

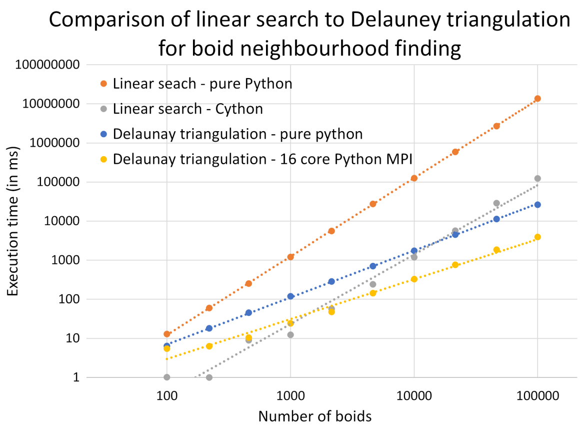

Figure 6: Comparison of the linear search algorithm and Delaunay triangulation algorithm for constructing the neighbourhoods of different numbers of boids. Running on up to 16 cores of the BC3 supercomputer.

Due to much higher complexity of the DT algorithm, for low numbers of boids it is no faster than the linear search. However, as expected, the DT algorithm execution time scales with the number of boids as $O(n\, log(n))$. Fitting a power law to the timing results, I found that below 15026 boids, the pure Python Delaunay triangulation algorithm is slower than the fastest Cythonised linear search (see figure 6).

The DT algorithm was parallelised using MPI. The set of boids was split evenly across the n CPU cores. Each core then triangulated its set of boids. A final step, run on a single core, then merged the n separate triangulations into the final result. Running the Delaunay triangulation on 16 cores using MPI, above 1465 boids the DT algorithm outperformed the Cythonised linear search.

At 100,000 boids the 16 core DT algorithm took 3.905s, compared to a python linear search time of 227 minutes; a factor of $\approx 3500\times$ speed-up. As the number of boids is increased, this factor grows arbitrarily large, as evident by the shallower gradient observed for the DT algorithm in figure 6. For further timings comparisons see table 3.

Having spent most of this paper addressing the problem of determining which boids to consider when implementing the boid rules, I now turn to implementing the boid rules themselves.

4 Boids Simulation

With the neighbours of each boid determined via the DT algorithm, the separation, cohesion and alignment rules are then applied. This updates the positions and velocities of each boid. By continually repeating the DT algorithm to find the new boid neighbourhoods and then applying the boids rules to update the positions and velocities, the full simulation is produced.

The cohesion rule requires the average position of the neighbours to be computed. This is achieved via the summation of the co-ordinates and then a division operation. The addition of several floating point values is computationally efficient for the CPU. Dedicated floating-point units (FPU) within the CPU are specially designed to carry out operations on floating-point numbers. Some modern processors can perform a floating point add operation in a single clock cycle. However, division operations can take 10 to 20 clock cycles [3]. To get around this bottleneck, I experimented with removing the division operation. Instead, I compared the sum of the n many neighbours positions to n times the test boid position. This replaces the 10–20 cycles needed for the division with 1–4 cycles needed for a floating point multiply. In theory I expected to see a significant speed-up. However, in Python no such difference was observed. It would be interesting to see if this proposition does provide a speed-up in a lower level implementation such as C++ or even assembly language.

I can justify focusing my efforts on the nearest neighbour search problem with the following statistic. When computing a single iteration of the Boids simulation, the percentage of the run-time taken up in updating the boids is $2.3 \pm 0.2\%$ (averaged across all numbers of boids). Hence, determining the neighbourhoods of each boid via the DT algorithm takes $97.7 \pm 0.2\%$ of the run-time. For 1000 boids, 160.3 ± 3.5ms to determine the boid neighbourhood and just 3.5 ± 0.35ms to update all the positions and velocities. At around 165 ms per iteration, this achieves roughly 6 frames per second. However, this is without any visualisation for the user to observe; simply the background computation.

5 Graphics

I initially used the matplotlib library to create an animation of the boids plot. However, plotting a single frame of the simulation took 100s of milliseconds (1.324 seconds for 1000 boids). This plotting time eclipses the 165 ms of run-time for a single time-step of the boids.

Instead, I created a custom plotting module based on the optimised computer vision library OpenCV. With a test setup of 1000 boids and a 1000x1000 pixel image my initial implementation took 52.56ms per frame, an order of magnitude better than matplotlib. With further improvements, such as creating my own colourmap class to replace the matplotlib colourmap module, I further reduced the plotting time to 15.64ms per frame. These optimisations improved the animation FPS from 0.75fps using matplotlib to 63.9fps using my OpenCV based module, an $85\times$ increase

Number of points	Linear search - 1 Core - Pure Python	Linear search - 1 Core - Cython	Delaunay triangulation - Pure Python	Delaunay triangulation - 4 core Python MPI	Delaunay triangulation - 16 core Python MPI
100	12.96	1.01	6.35	4.50	5.50
1000	1212.19	12.36	119.00	55.24	24.06
10000	124972.74	1183.48	1740.00	899.56	329.82
100000	13626878.87	124589.00	26307.00	12372.50	3904.58

Table 3: Table of execution times (in ms) comparing the linear search algorithm with the Delaunay triangulation algorithm to compute the neighbourhoods of different numbers of points ranging from 100 to 100,000.

in FPS. An example frame of the animation is shown in figure 7.

6 Conclusion

To run ever more complex simulations, computational physicists employ massively parallel systems to run code concurrently. OpenMP and MPI are two of the most widely recognised programming frameworks for parallel computing. In this report I have demonstrated the performance improvements available using these frameworks via a flocking swarm simulation.

I found that using OpenMP, every factor of 2 increase in the thread count reduced the runtime by $1.95 \pm 0.05\times$. With MPI I measured this factor to be $1.7 \pm 0.2\times$ with each doubling of the core count.

I employed Cython to produce precompiled code, and found that my linear search nearest neighbour algorithm was $104 \pm 8\times$ faster than the pure Python code using Cython.

I then reduced the computational complexity of my code by employing a Delaunay triangulation algorithm to compute the nearest neighbours in my Boids simulation. I investigated a range of python data structures to implement this. The Delaunay triangulation algorithm was then parallelised using MPI.

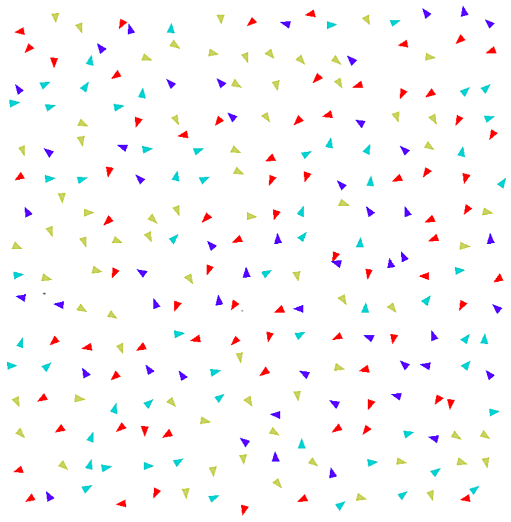

Figure 7: Plot showing the first frame of the Boids animation for 256 boids. The boids start on a grid/lattice distribution with some small random offset. The directions of travel are uniformly distributed, with a colour-map of four colours representing each quadrant of travel direction.

In writing my own optimised plotting module, I improved the animation FPS from 0.75fps using matplotlib to 63.9fps using my OpenCV based module.

There are many ways I would like to improve this project. One example would be to utilise the CUDA (compute unified device architecture) model for parallel computing on GPUs. An interesting comparison would be to see if my parallel Delaunay triangulation approach to the Boids simulation could be implemented using CUDA for massively parallel execution on a GPU.

References

[1] Craig W. Reynolds. 'Flocks, Herds, and Schools: A Distributed Behavioral Model'. In: *SIGGRAPH Computer Graphics* 21.4 (July 1987), pp. 25–34. DOI: 10.1145/37402.37406.

[2] Marcello Pelillo. 'Alhazen and the nearest neighbor rule'. In: *Pattern Recognition Letters* 38 (Mar. 2014), pp. 34–37. DOI: 10.1016/j.patrec.2013.10.022.

[3] Victor Eijkhout. 'Introduction to High Performance Scientific Computing'. In: (2012).

[4] François Broquedis et al. 'ForestGOMP: An Efficient OpenMP Environment for NUMA Architectures.' In: *Int J Parallel Prog* 38 (2010), pp. 418–439. DOI: 10.1007/s10766-010-0136-3.

[5] Sol Ji Kang, Sang Yeon Lee and Keon Myung Lee. 'Performance Comparison of OpenMP, MPI, and MapReduce in Practical Problems'. In: *Advances in Multimedia* 2015 (Aug. 2015), p. 575687. ISSN: 1687-5680. DOI: 10.1155/2015/575687. URL: https://doi.org/10.1155/2015/575687.

[6] Boris Delaunay. 'Sur la sphère vide'. In: *Otdelenie Matematicheskikh i Estestvennykh Nauk* 7 (1934), pp. 793–800.

[7] Jonathan Richard Shewchuk. 'Delaunay refinement algorithms for triangular mesh generation'. In: *Computational Geometry* 22.1 (2002). 16th ACM Symposium on Computational Geometry, pp. 21–74. DOI: 10.1016/S0925-7721(01)00047-5.

[8] Jonathan Richard Shewchuk Carnegie Mellon School of Computer Science. *Triangulation Algorithms and Data Structures*. 1996. URL: https://www.cs.cmu.edu/~quake/tripaper/triangle2.html (visited on 28/02/2021).

[9] Leonidas Guibas and Jorge Stolfi. 'Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi'. In: *ACM Trans. Graph.* 4.2 (Apr. 1985), pp. 74–123. ISSN: 0730-0301. DOI: 10.1145/282918.282923.

[10] Samuel Peterson University of Minnesota. *Computing Constrained Delaunay Triangulations*. 1997. URL: http://www.geom.uiuc.edu/~samuelp/del_project.html (visited on 28/02/2021).

[11] Russel Ahmed Apu and Marina L. Gavrilova. 'Efficient Swarm Neighborhood Management Using the Layered Delaunay Triangulation'. In: *Generalized Voronoi Diagram: A Geometry-Based Approach to Computational Intelligence*. Ed. by Marina L. Gavrilova. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 109–129. ISBN: 978-3-540-85126-4. DOI: 10.1007/978-3-540-85126-4_5. URL: https://doi.org/10.1007/978-3-540-85126-4_5.