Boids: A Parallel Approach

Magnus Ross

Department of Physics,
University of Bristol.

Boids simulations are a way to model the collective behaviours of large groups of individual actors (or 'boids'), for example a shoal of fish or a murmaration of starlings. The beauty of a boids simulation is that these collective behaviours emerge from a small set of simple, and most importantly *local* rules, as they would in nature. The Boids algorithm was developed by Craig Reynolds in 1987 (1). Many different rules can be applied to the boids to encourage different behaviours. In this project, however, the three original rules will be adopted: Collision Avoidance, Velocity Matching, and Flock Centering.

The goal of this project is to develop a parallel boids simulation that is both fast and efficient and that scales well with increasing data size. This algorithm should effectively leverage the huge computing power provided by the University of Bristol's HPC system, Blue Crystal 3 (BC3). Over the course of this project, three distinct but related parallel boids algorithms have been developed, each with advantages and drawbacks. In this report I will explain these algorithms and analyse their speed, efficiency and complexity.

Section 1 will contain details of the algorithms and the frameworks used for their implementation, as well as a brief analysis of the theoretical complexity of each algorithm. Section 2 will contain the results of practical application of the algorithms on BC3. Section 3 will contain some outlook and suggestions for future directions.

I. IMPLEMENTATION

A. MPI

Open Message Passing Interface (MPI) is an open source framework for parallel programming on distributed memory computing systems (2). MPI has been implemented in a variety of languages including C and FORTRAN. In this project the Python version is used, which is known as `mpi4py` (3). MPI was chosen for this project because it is designed for distributed memory systems. The maximum size of a group of processors with shared memory on BC3, one node, is 16. In one wishes to use more processors than 16 then a distributed memory framework is needed. One possibility is using both distributed and shared memory frameworks in tandem, to exploit the advantages of each, this is not done here for reasons that will be discussed shortly. MPI is a mature tool that has been developed for over 20 years and is compatible with a large range of different computing systems, which is another reason it has been chosen for this project.

B. Numba

Python is an interpreted, dynamically typed language, these features offer many benefits, and typically Python programs are pleasant to read and write when compared to a statically typed, complied languages like C or FORTRAN. The dynamically typed and interpreted nature of Python does however mean it is considerably slower than languages like C and FORTRAN, sometimes 2-3 orders of magnitude slower. This is clearly unacceptable in scientific computing contexts, where speed is essential. Many efforts have been made to improve Python's speed for scientific computing tasks. The most successful of which is probably the NumPy package, which brings highly optimised array based operations, sometimes giving 3 orders of magnitude speedup over pure Python (4). NumPy is now ubiquitous in the scientific computing and machine learning communities. Another option is Cython, which complies Python to C code.

Numba is a package for Python which allows functions to be just-in-time (JIT) complied to fast machine code (5). When correctly used, Numba can offer huge speedups of roughly 3 orders of magnitude, and allow functions written in pure Python to compete with C or FORTRAN in terms of speed. In theory Numba is the best of both worlds, offering the simplicity and readability of pure Python, with the speed of a compiled language. In practice, however, programming with Numba can be very frustrating. This is because the error messaging often offers little information when functions fail to compile. In addition to this there are a number of Python features that are not supported by Numba. The documentation and error messaging for Numba has significantly improved in recent years however, and new Python features are being added all the time, one notable recent example being classes. Numba offers a variety of other important features including support for GPU programming, easy parallel programming, and vectorised instruction sets, like AVX. Not all of these features can be used with BC3 however. For example, auto parrallelisation cannot be used. In this project a combination of Numba, Numpy and MPI are used.

C. Algorithms

The boids update algorithm will not be explicity written here, since it is widely available online. A clear and exhaustive explanation is given in (6), which was used to write the code for this project. From (6) the rules 1, 2 and 3 were used, corresponding to Collision Avoidance, Velocity Matching, and Flock Centring from (1). The pseudo code for the three parallel algorithms developed as part of the project is listed in below.

1. Basic

Algorithm 1 will subsequently be referred to as Basic. This algorithm is the most simple parallel implementation of the boids simulation. It does not capitalise on the spatial distribution of the boids and simply divides the work required to update the boids equally between the processors. The algorithm has a time complexity of $\mathcal{O}(N_B^2)$ where N_B is the number of boids. Since the work is divided equally among the processors, the parallel part of the algorithm should see a speed up of Q, where Q is the number of workers [1]. Each iteration of the algorithm involves sending a message of size $\mathcal{O}(N_B)$, $4Q$ times [2].

Algorithm 1 Basic

```
if Master then
    Initialise positions and velocities
    Assign boids to workers
    Send assignments to workers
    for i = 1, 2, ..., N_it do
        Broadcast positions and velocities;
        Wait
        for i = 1, 2, ..., Q do
            Receive updated positions and velocity
        end for
        Collate updates
        Broadcast updated positions and velocities
    end for
else
    Receive boid index assignments
    for i = 1, 2, ..., N_it do
        Receive updated positions and velocities
        Calculate position and velocity updates for assigned boids
        Send updates for assigned boids to master
    end for
end if
```

[1] The total number of processors required to run simulation Q workers for all algorithms is $Q + 1$, since all have an additional Master that does no computation.

[2] The message numbers are calculated from the actual implementation, not from the pseudo code here. Additional single integer messages are also sent, but these are always sent immediately before an array message so are not included.

2. Grid

Algorithm 2 will subsequently be referred to as Grid. This algorithm uses assigns each worker to a spatial region (cell) of the problem domain. When a boid enters each cell it is then owned by that worker and that worker performs the updates on the boid. When updating the boids, workers only consider the boids in cells that are adjacent to them. This clearly saves a large amount of computation. This means that the size of each cell must be greater than the field of view of the boids. Additionally since in my implementation the domain is divided into regular squares, only square or cube numbers of processors can be used in 2D or 3D simulations. Both of these factors are major disadvantages to the grid method. The time complexity of this method is still $\mathcal{O}(N_B^2)$ however the coefficient of N_B^2 will generally be smaller, although this is dependent on how the boids move around the domain. In the best case where the boids are evenly distributed across the domain the speed up of the parallel part is Q. However in the worst case where the boids all coalesce into a small region, this algorithm provides no speed up, since one worker is doing all the computation. The number of messages is also much larger, with $10Q$ and $28Q$ messages of size $\mathcal{O}(N_B/Q)$ being send in 2D and 3D simulations. The Grid algorithm is worse by almost every measure than the other two algorithms, however it was a step in the development of the Balanced algorithm, which combines the best elements of Grid and Basic, so is included here.

Algorithm 2 Grid

```
if Master then
    Initialise positions and velocities
    Assign workers to cell of spatial grid of domain
    Send assignments to workers
    for i = 1, 2, ..., N_it do
        Determine which cell each boid is in
        for i = 1, 2, ..., Q do
            Send each worker the boid it owns
            Wait
            Receive updated positions and velocity
        end for
        Collate updates
    end for
else
    Determine Neighbours
    Receive spatial assignment
    for i = 1, 2, ..., N_it do
        Receive assigned boids positions and velocities
        for i = 1, 2, ..., N_Neigh do
            Send boids positions and velocities to neighbour
            Wait
            Receive positions and velocities from neighbour
        end for
        Calculate updates using own and neighbours boids
        Send updates to master
    end for
end if
```

3. Balanced

Algorithm 3 will subsequently be referred to as Balanced. In Balanced, workers are assigned specific boids to manage, similar to Basic. A spatial grid is also maintained with a list of which boid is in which grid. When a boid is to be updated, the worker only needs to query the grid list to see what cell the boid is in, and then calculate updates only with respect to the neighbouring grid cells. Since the grid structure is independent of the workers the load can be equally balanced between the workers, whilst still keeping the benefit of only calculating updates with respect to local boids. This independence also means that the grid size can be set to exactly the field of view of the boids, so the minimum number of boids are checked against. In the limit of large field of view (so large grid squares), this algorithm actually becomes the same as Basic. Balanced has a time complexity of $\mathcal{O}(N_B^2)$ with the coefficient of N_B^2 proportional to the field of view. In the limit of small field of view the time complexity becomes $\mathcal{O}(N_B)$ since there is no interaction between boids, ie no boids in neighbouring grid cells. The speedup is not limited by the spatial distribution of boids as in Grid so Q workers provide Q speedup to the parallel part. In the balanced algorithm all computation, including updates to the grid list, are done by the workers, master is only used for amalgamation of updates and message sending. This limits the length of the serial part of the program. Each iteration involves sending $5Q$ messages size $\mathcal{O}(N_B)$ and $2Q$ smaller messages of variable size which contain the gird updates.

Algorithm 3 Balanced

```
if Master then
    Initialise positions and velocities
    Divide space up into cells, size of squares should be boids field of view
    Make fixed assignment of boids to workers
    Send assignments to workers
    Determine which grid cell each boid is in
    for i = 1, 2, ..., N_it do
        Broadcast positions and velocities
        Broadcast which boid is in which cell
        Wait
        for i = 1, 2, ..., Q do
            Receive updated positions and velocity
            Receive labels of boids who have moved cell, and where they are now
        end for
        Collate updates
        Broadcast updated positions and velocities
    end for
else
    Receive boid index assignments
    for i = 1, 2, ..., N_it do
        Receive updated positions and velocities
        Recieve which boids is in which cell
        Calculate position and velocity updated for assigned boids
        Calculate which of assigned boids have moved cell
        Send updated position and velocity for assigned boids to master
        Send cell updates to master
    end for
end if
```

II. RESULTS

In Figure 1 a frame from the simulation of 500 boids in 2D space exibiting flocking behaviour is shown. All the results in the section are in 2D space, however the code does also support 3D simulation [3]. In general the dimension does not make a difference to the conclusions drawn from the results. The only slight difference is a marginally worse performance for all algorithms, and a marginal drop in relative performance for the Balanced and Grid algorithms when compared to the Basic algorithm. This is because in 2D there are 8 neighbours for each cell, whereas in 3D there are 26. All the results here utilise the Numba package unless otherwise stated. In all simulations the positions of the boids are 'wrapped', such that when a boid goes over the domain boundary, it appears on the opposite boundary and comes back into the domain. In the simulation here boids do not feel forces through the domain boundaries. This behaviour could be added to make the simulation more realistic, but was not considered here due to the limited time available and the fact it is not related to the parallelisation of the code.

[3] Videos of both 2D and 3D simulations can be found on the GitHub repository for this project at `https://github.com/magnusross/Adv-Comp`

A. N_B Scaling

In Figure 2 the scaling of each algorithm as the number of boids, N_B, is increased is shown. Here a square domain of size 1000 was used with a field of view of 50. The simulations were run on BC3 using 26 processors ($Q = 25$), with 3 repeats. All the algorithms show clear N_B^2 scaling, with all fits having and $r^2 > 0.999$, where r^2 is the proportion of explained variance. As expected the coefficient of N_B^2 is considerably smaller for Balanced when compared to the the others, approximately 3x smaller than Basic and 5x smaller than Grid. The performance of Grid is poor relative to the other algorithms, as the uneven distribution of flocking boids means that it cannot capitalise on the additional processors.

B. Field of View Scaling

As briefly mentioned in Section I.C.3, the performance of the Balanced algorithm should decrease as the field of

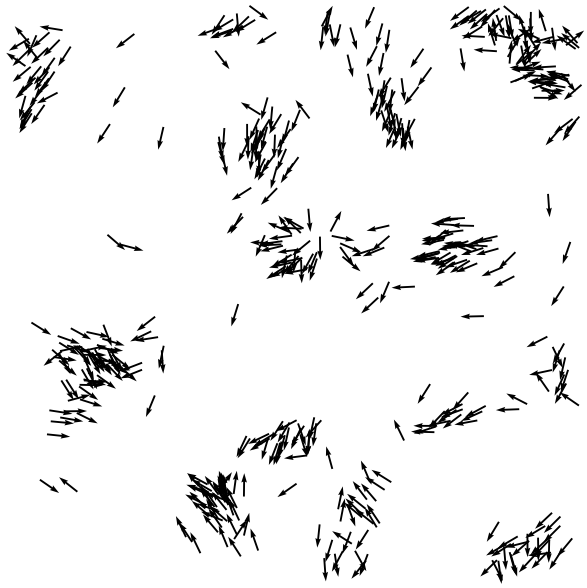

FIG. 1 A graphic of the one frame of the boids simulation in 2D, with collective flocking behaviour being exhibited. Arrows here show the direction the boids are travelling in, but not the velocity, since they have all been normalised. The Balanced algorithm was used to generate this graphic.

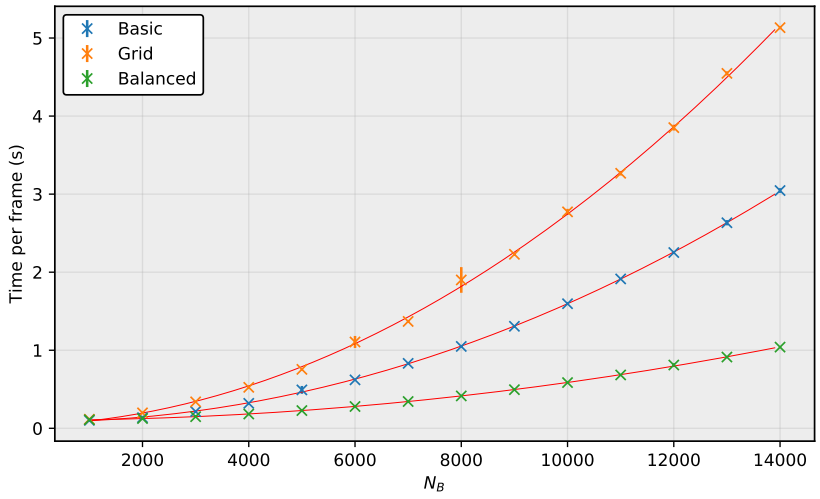

FIG. 2 A graph of the scaling of the three algorithms, vertical axis showing time per frame, horizontal axis showing number of boids. There are error bars on the graphs however in most cases they are too small to be seen. Red lines shows least squares fit to second degree polynomial

view is increased. More concretely, the coefficient of N_B^2 should grow as the field of view does, since each neighbouring cell contains more boids. Figure 3 shows the scaling with increasing N_B for Balanced, with 3 different fields of view, as well as Basic. For all methods a square domain side length 1000 was used, and was run locally using 4 processors ($Q = 3$), with 3 repeats. For the field of view 450 we would expect the Balanced algorithm to have the same performance as Basic, because only 4 cells would be created, meaning every boid has to calculate updates with respect to every other boids, since every cell is adjacent to every other cell. We can indeed see that this is the case, with the red and blue fits being almost identical, this slight difference in shape coming from details of the implementation. The r^2 values of the fit were 0.9995, 0.9992, 0.9943 and 0.9515 for Basic and

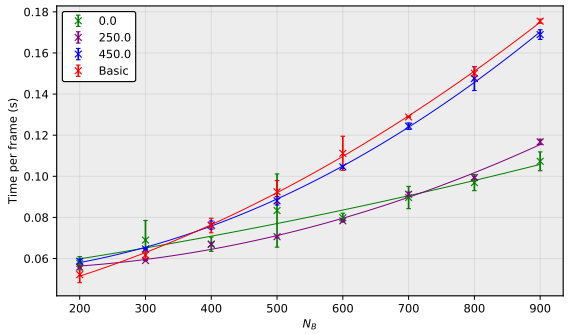

FIG. 3 A graph of the scaling of the Balanced method as field of view is varied. Vertical axis showing time per frame, horizontal axis showing number of boids Smooth lines show second degree polynomial fit.

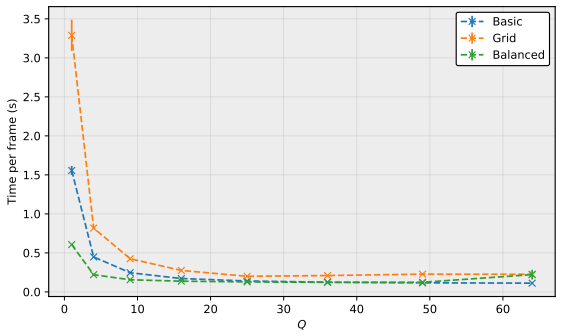

FIG. 4 A graph showing execution time when the number of processors is increased. Horizontal axis shows number of processors, vertical time per frame. Error bars are shown but most are too small to see. Dotted lines are not fits, they are just there as a visual aid.

fields of view 450, 250 and 0 respectively.

We can clearly see that the trend for field of view 0 is linear as mentioned in Section I.C.3. r^2 values all data sets with a linear fit were also calculated. The only data set which had a greater value for the linear fit than the second degree polynomial fit was field of view 0. This supports the claim of linear scaling when the field of view is 0.

C. Q Scaling

In Figure 4 the effect of increasing the number of workers, Q, on the running time of the algorithms can be seen. The data was generated using 2000 boids, a square domain size 1000 and field of view 50, with 3 repeats, all on BC3. In Figure 5 the same data is plotted, but with speedup on the vertical axis instead of time. Speedup is defined here as

$$S = \frac{T_1}{T_Q} \tag{1}$$

where T_Q is execution time for Q processors.[4] All methods exhibit Amdahl's law to varying extents. Amdahl's law is given by

$$S = \frac{1}{(1-P) + (\frac{P}{Q})} \tag{2}$$

where S is the speedup given by using Q processors, and P is the fraction of the program that can be parrallelised (7). The data in figure 5 was fitted to equation 2, including uncertainties [5], using P as the free parmeter. This gave values for P of $94.68 \pm 0.06\%$, $97.0 \pm 0.4\%$ and $82.8 \pm 0.2\%$ for Basic, Grid and Balanced respectively. It should be noted that the value of P will be dependent on N_B, with larger N_B meaning more of the time the program spends executing is in parallel. More large messages are sent between master and workers in Balanced [6], additionally the collation of the grid updates must be done on master so one would expect this method to have the largest serial proportion, as confirmed by the values above. The Grid method is least well fitted, which would be expected since its speedup depends on the spatial position of the boids in a complex way. The Basic method is most well fitted by far, this follows because it is a simple algorithm, and the speed is independent of the spatial position of the boids. The uncertainties for all methods become large at high Q. This could be due to communication latency on BC3, which is more acute when a large number of separate nodes are being used, which is required when many workers are requested.

D. Numba

In Figure 6, a graph showing the speedup provided by the Numba package can be seen. Here the Balanced method was used with 4 processors, a square domain size 500 and a field of view 100. We can clearly see that the Numba package provides a performance boost for any size greater more boids than about 200. The computational part of Numba is so fast that barely any scaling can be seen when increasing the number of boids. There is large amount of overhead that comes from using Numba with MPI. When a Numba function is called it must be compiled, usually this only happens the first time it is called

[4] This ignores the additional processor that acts as Master

[5] Propagated from data in Figure 4 using $\Delta S = T_1 \frac{\Delta T_Q}{T_Q^2}$

[6] Although more messages total are sent in Grid, these are sent between workers

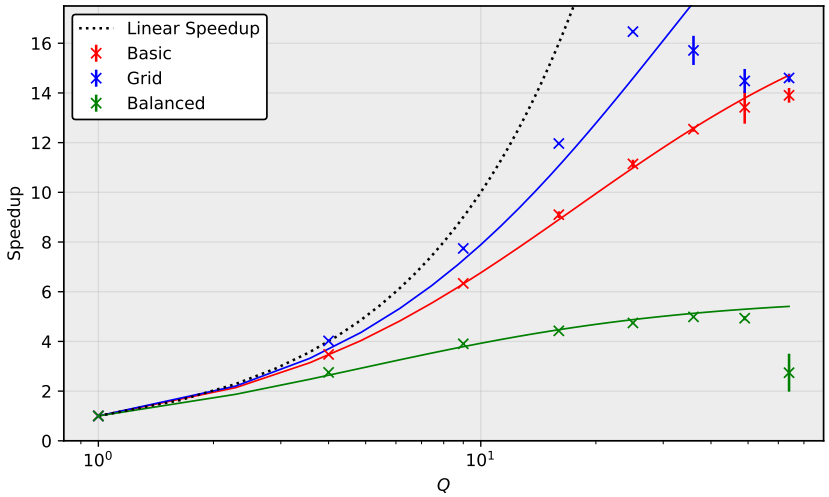

FIG. 5 A graph speedup against number of processors. Horizontal axis is logarithmic. Solid lines show show fits to equation 2 with P as a free parameter. Dotted line shows perfect linear speedup for comparison.

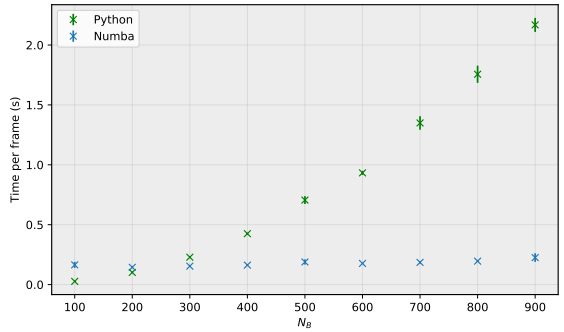

FIG. 6 A graph showing the execution time of the Numba compiled functions when compared to normal Python.

and then the compiled version is cached for later use, so it does not have to be complied again. When a new MPI process is spawned it cannot access the compiled version of the function, and so must recompile it. This effect is only significant at very small data sizes, since when the data is bigger, the vast majority of the time is spent on computation not the compilation. One could eliminate this problem by using the a feature in Numba that allows pre-compilation of functions to `.so` files, getting rid of the compilation overhead. This also allows pre-compiled functions to be used on systems that do not have Numba installed.

III. DISCUSSION

There are many possibilities for improving the methods presented here, both in an algorithmic sense and in the details of the implementation. Here only distributed memory parallelism has been used. Although distributed memory systems are able to ultimately access more workers than shared memory systems, they are naturally slower, since large messages containing data must be sent between iterations. For distributed memory to be a favourable choice, the algorithm must be able to put the additional workers to good use. This is not really the case for my fastest algorithms since the speedups are limited by Amdahl's law at quite low Q. For this reason, implementing using a shared memory framework, like OpenMP, may have provided a faster solution. It would have been relatively easy to exploit shared memory parallelism within my method, by using the Numba `prange` feature, however this was not supported on BC3.

GPUs are also a promising option for boids simulations. GPUs contain a large number of small processing cores, with fast shared memory. An implementation of the boids simulation using CUDA on a GPU was able to achieve simulation of 5000000 at 11 frames per second (8). This constitutes approximately a 1000x speedup on the methods presented here, an impressive number.

Algorithmic improvements could be made by using more complex data structures, for example quad-trees and oct-trees. This would allow communication to be managed more efficiently. A parallel boids simulation using oct-trees was implented in (9), however the age of this work means speeds cannot be fairly compared.

REFERENCES

[1] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th*

annual conference on Computer graphics and interactive techniques, pages 25–34, 1987.

[2] https://www.open-mpi.org.

[3] https://mpi4py.readthedocs.io/.

[4] https://numpy.org.

[5] http://numba.pydata.org.

[6] Conrad Parker. http://www.vergenet.net/~conrad/boids/pseudocode.html.

[7] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery. doi:10.1145/1465482.1465560.

[8] Aman Scahan. CUDA Boid Flocking (GitHub repository). https://github.com/AmanSachan1/CUDA-Boid-Flocking, 2017.

[9] Thomas Portegys and Kevin Greenan. Managing flocking objects with an octree spanning a parallel message-passing computer cluster. pages 683–687, 01 2003.