Investigating a parallelised boids simulation

David Tudor

November 2024

Abstract

A boids simulation was parallelised using shared and distributed memory and the results of the two methods were compared. The programs were run on BlueCrystal4 and used up to 28 threads which gave maximum speed-ups of 17.02 and 12.32 using OpenMP and MPI respectively for $250,000$ boids. As the number of boids was increased, the computation time increased slower than quadratically due to other parts of the program like the parallelisation becoming more efficient for larger system sizes.

1 Introduction

The purpose of this experiment is to investigate how the computation time of a boids simulation scales with the number of simulated boids and the number of threads used. Shared and distributed memory methods are implemented to parallelise the program using OpenMP and Message Passing Interface (MPI) respectively and the results of the two are compared. A boids simulation simulates the flocking of animals like birds [1].

For shared memory with OpenMP, there is a block of memory which can be accessed by all threads [2] allowing any thread to quickly access the result of a calculation completed by a different thread. However, this memory needs to be managed to keep it synchronised and ensure two threads don't write to the same location at the same time.

In contrast for distributed memory with MPI, all threads (also called ranks) have separate memories so must communicate by sending data to each other [3] which introduces more overhead. This communication is slower than using shared memory but can be used between nodes such as on BlueCrystal4 (BC4) instead of just between cores on a node like shared memory. The task must be explicitly divided into sections for each rank to work on. Unlike with shared memory, each rank runs the entire program but conditionals can be used to make them run different parts of the code.

The speed-up S_N and efficiency E_N are used to quantify the effectiveness of using more threads [4] and are calculated using the computation time on a single thread T_1 and the time on N threads T_N. Their formulae are given in equations (1) and (2).

$$S_N = \frac{T_1}{T_N} \tag{1}$$

$$E_N = \frac{S_N}{N} \tag{2}$$

The speed-up will always asymptote towards some value due to the computation time of the serial code. Amdahl's law [5], equation (3), relates the maximum speed-up S_N to the number of threads N and the fraction of parallel code f.

$$S_N = \frac{1}{(1-f) + \frac{f}{m}} \tag{3}$$

2 Method

2.1 Boids simulation

Each boid is affected by three factors which cause it to steer in different directions [1]. Alignment causes it to steer to match the velocity of other nearby boids. Cohesion causes it to steer towards the average position of nearby boids whilst separation causes it to steer away from the average position of the boids which are too close. These rules result in the boids travelling in flocks without colliding. Equation (4) shows how this is implemented and the coefficients A, C and S can be used to give different weightings to the alignment, cohesion and separation terms respectively. The nearby boids are the ones within some vision radius and the close boids are the ones within a smaller radius; N_{nearby} and N_{close} are the numbers of nearby and close boids respectively. The alignment term takes the average velocity difference between all the nearby boids and the i^{th} boid; the cohesion term does the same but uses the average position difference. The separation term has the opposite sign so the i^{th} boid steers away from boids which are too close. The $|\mathbf{r}_j - \mathbf{r}_i|^2$ denominator weights this heavily to avoid the closest boids. As well as steering, equation (4) also results in the speed changing so all the speeds were reset to a constant value after each time step. The Euler method was used to time step the system, as shown in equations (4) and (5), instead of a more accurate method like the fourth order Runge-Kutta method because the system does not model a specific real-world example. For example, the coefficients of the three factors were adjusted until the expected flocking behaviour was seen rather than being set through experimental observation. The boundaries of the bounding box were made to be periodic and differences in position accounted for this such that two boids near opposite edges could be close. All simulations were done in two dimensions.

$$\mathbf{v}_i = \mathbf{v}_i + \left(A \cdot \frac{\sum_{j=1}^{N_{nearby}} \mathbf{v}_j - \mathbf{v}_i}{N_{nearby}} + C \cdot \frac{\sum_{j=1}^{N_{nearby}} \mathbf{r}_j - \mathbf{r}_i}{N_{nearby}} - S \cdot \frac{\sum_{j=1}^{N_{close}} \frac{\mathbf{r}_j - \mathbf{r}_i}{|\mathbf{r}_j - \mathbf{r}_i|^2}}{N_{close}} \right) \cdot dt \tag{4}$$

$$\mathbf{r}_i = \mathbf{r}_i + \mathbf{v}_i \cdot dt \tag{5}$$

To accelerate identifying the nearby boids, the bounding box was divided into a grid of cells with each cell storing its contained boids. The side length of each square grid cell was set to the vision radius or greater. Therefore, only the boids in the three-by-three square of grid cells around any boid's cell needed to be checked as any others would be too far away. This is demonstrated by the shaded region in Figure 1 and the red circle shows a particular boid's vision radius. For efficiency, the search result of which boids were in the surrounding cells was reused for every boid in the central cell.

The state of the system was recorded at the end of each time step, stored in a $.npz$ file and plotted after the simulation as an animation using $matplotlib.animation$. Some example frames are shown in Figure 2.

To complicate the system, the boids were given a field of view of 4.8 radians to stop them from seeing backwards. This prevented a potential optimisation from using one check to find if two boids could see each other. Instead, two vision checks are required as the angle checks are non-symmetric.

To prevent the boids from eventually forming one large flock, hawks were added which chase the nearest boid and are slightly faster than them. When they get close enough, they eat the boid and

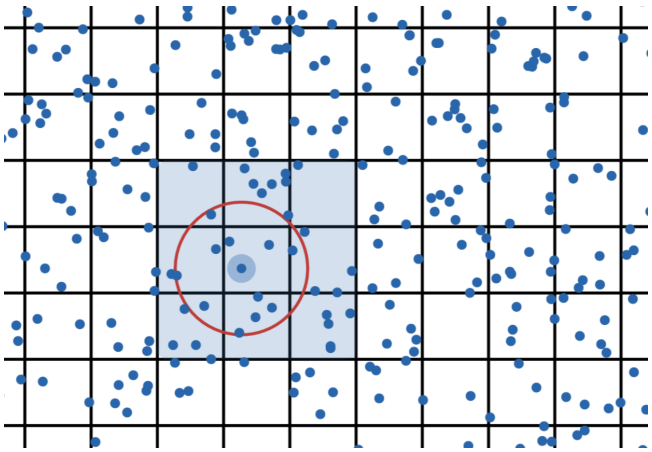
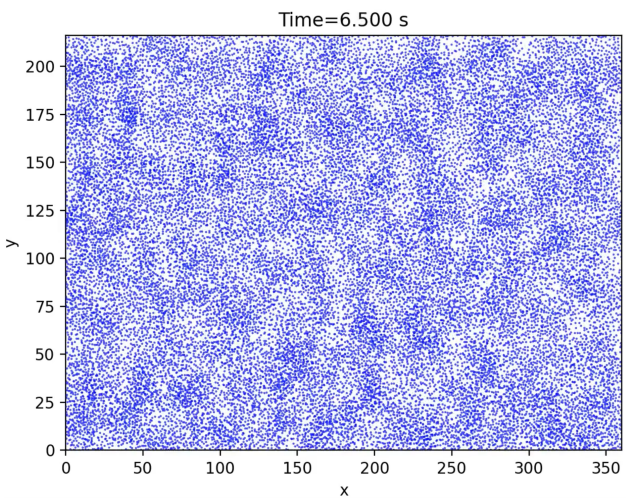
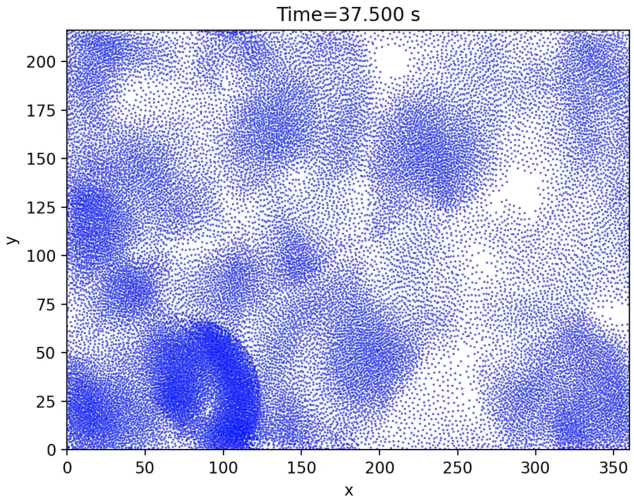


Figure 1: Diagram showing the number of vision checks can be filtered to the number of boids in the shaded region using the grid. The red circle shows the vision area of the boid.



(a) State near the start of the simulation.



(b) State after some time.

Figure 2: Frames taken from an animation with $30,000$ boids. Each blue dot is a boid.

stop it from moving and interacting further. The boids were made to flee from the average position of the hawks that they could see and to prioritise this over flocking. Figure 3 shows an example of this and shows pockets form around the hawks as all the boids fly away from them.

2.2 OpenMP

OpenMP was used to parallelise part of the program. Each time step, the grid was calculated on a single core using NumPy. Then, a $prange$ was used to iterate over all the grid cells in parallel. This was much faster than when the $prange$ was used to iterate over the boids in each grid cell which required starting a new $prange$ for each grid cell and led to the program slowing down when more than four threads were used.

2.3 MPI

For MPI, a manager rank was used to initialise the simulation and $Bcast$ was used to broadcast the current state of the system to all ranks at the start of each time step. Using this data, the grid cells

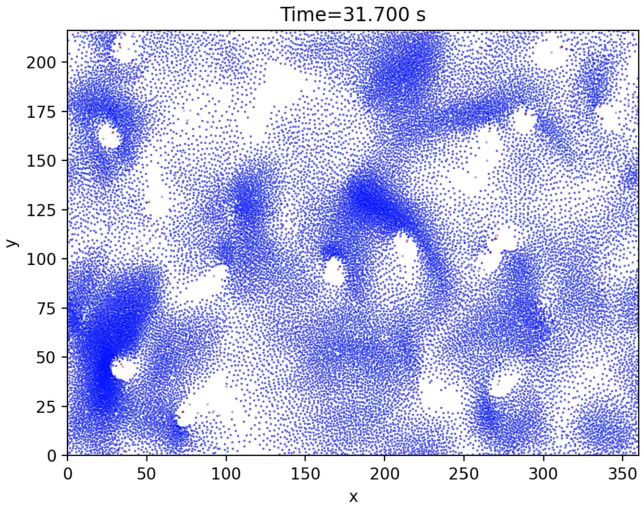


Figure 3: Frame from an animation with $30,000$ boids and 15 hawks. Blue dot are boids, red dots are hawks, grey dots are eaten boids.

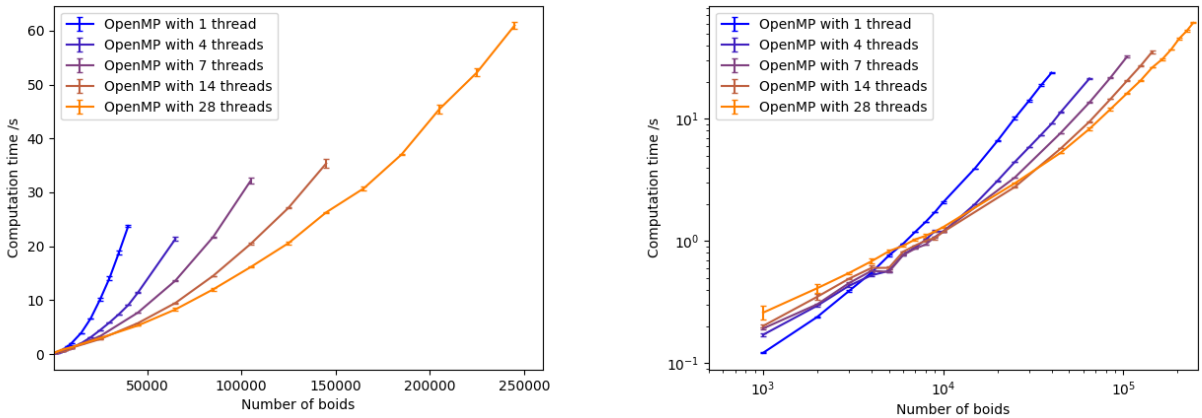
were allocated sequentially to the rank with the least current work. The current work was defined by the total number of boids in all the cells already allocated to each rank. Each rank calculated the allocation individually to reduce the number of broadcasts needed. The manager rank was also allocated work since communication was not required except at the start and end of the time step. Because each rank was allocated a different number of boids to calculate, $Gatherv$ was used to collect the calculated data in the manager rank. $Bcast$ and $Gatherv$ were used for communication between ranks as they use techniques including tree-based algorithms to make them faster than the simpler functions, $Send$ and $Recv$. NumPy arrays were sent between ranks rather than Python arrays as they are stored more efficiently.

3 Results

All timings were the average of three repeats simulated for 100 steps of 0.15 seconds and the error bars plotted show the sample standard deviation of these repeats. Boid and hawk initial positions and velocities were randomly generated. Unless specified otherwise, the simulations were run on BC4 with no hawks, the grid size was equal to the boid vision radius and dynamic scheduling was used for OpenMP's $prange$.

3.1 Scaling with system size

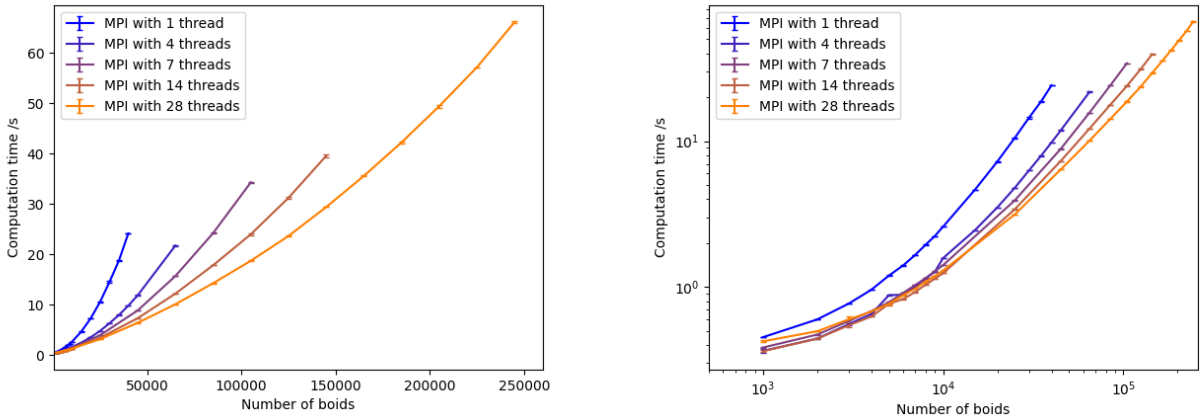
Figures 4 and 5 show how the computation time scales with the system size using OpenMP and MPI respectively. On the plots with logarithmic axes, the gradients of the OpenMP and MPI single thread lines were 1.85 and 1.68, whilst the 28 thread lines had gradients of 1.50 and 1.42. The last six points of the single thread lines and the last ten points of the 28 thread lines were used to calculate the gradients because the computation times of the smaller systems were dominated by the serial parts of the code rather than the calculation of the boid interactions.



(a) Linear axes

(b) Log axes

Figure 4: Computation time against the number of boids calculated on BC4 using OpenMP.



(a) Linear axes

(b) Log axes

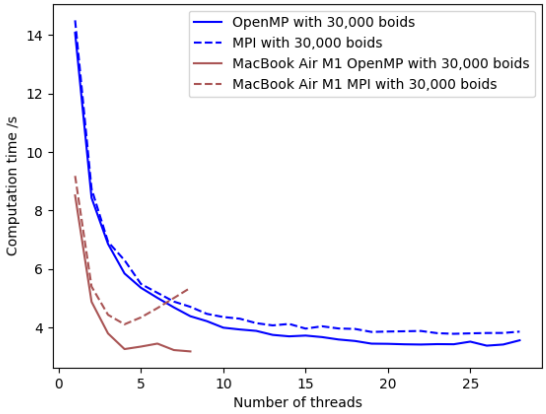
Figure 5: Computation time against the number of boids calculated on BC4 using MPI.

3.2 Scaling with number of threads

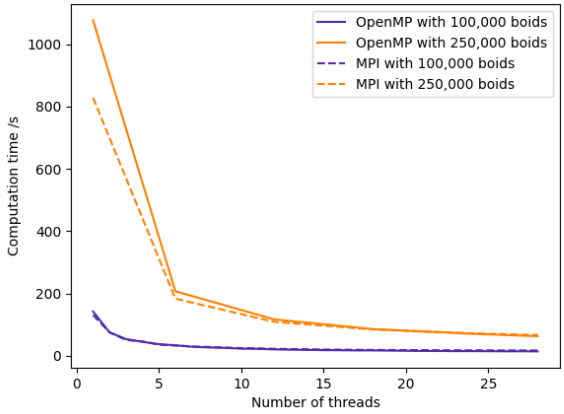
Figure 6a plots how the computation time scales with the number of threads with $30,000$ boids using OpenMP and MPI when run on BC4 and a MacBook Air M1. The time to compute larger systems of $100,000$ and $250,000$ boids was measured against the number of threads on BC4 using OpenMP and MPI and is shown in Figure 6b. Figures 6c and 6d plot the speed-up and efficiency of the data in the other two figures.

3.3 Tuning

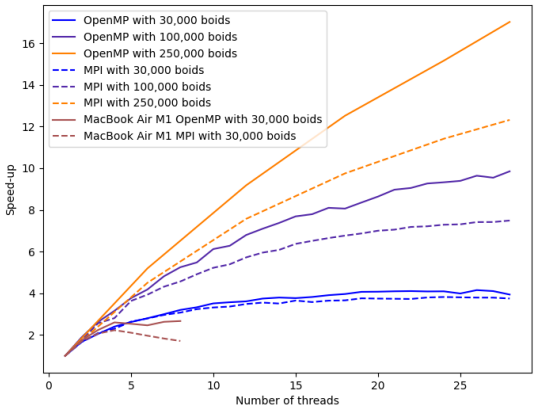
Different methods of delegating work to each thread were experimented with for two and eight threads. Figure 7 shows how changing the scheduling of the $prange$ affects the computation time using OpenMP for a range of system sizes. Figure 8 plots the computation time against the system size for three different methods of manual allocation using MPI. The first method was the one used for every other MPI graph and it allocates each grid cell to the rank with the least current work - this was quantified by the total number of boids in the grid cells already allocated to that rank. The second method was



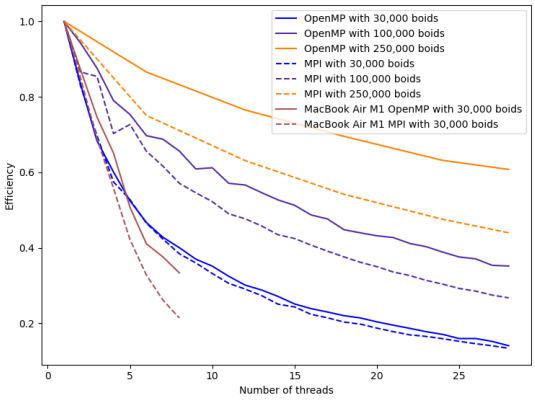
(a) Computation time against number of threads for $30,000$ boids run on BC4 and a MacBook Air M1.



(b) Computation time against number of threads for $100,000$ and $250,000$ boids run on BC4.



(c) Speed-up of Figures 6a and 6b.



(d) Efficiency of Figures 6a and 6b.

Figure 6: Scaling of the computation time against number of threads for various simulations. Solid lines used OpenMP and dashed lines used MPI.

the same but allocated the grid cells containing the most boids first. The third method was simpler and allocated the grid cells to each rank in a cycle using a modulus.

Figures 9 and 10 show how tuning the size of the grid cells and the number of hawks affects the computation time using OpenMP and MPI with four threads. The grid sizes tested were factors of the bounding box dimensions so that the box didn't need resizing.

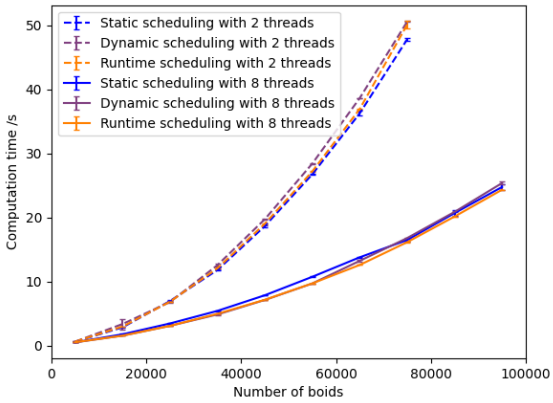


Figure 7: Computation time on BC4 against number of boids using static, dynamic and runtime scheduling in OpenMP.

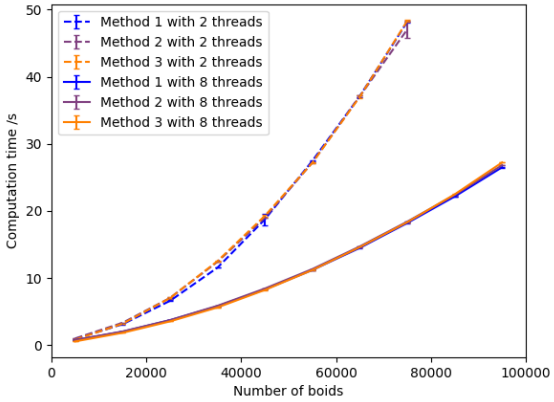


Figure 8: Computation time on BC4 against number of boids using MPI and different methods of allocating work to ranks. Method 1 allocates grid cells one by one to the rank with the least work. Method 2 allocates grid cells ordered from highest to lowest work to the rank with the least work. Method 3 allocates grid cells to ranks cyclically.

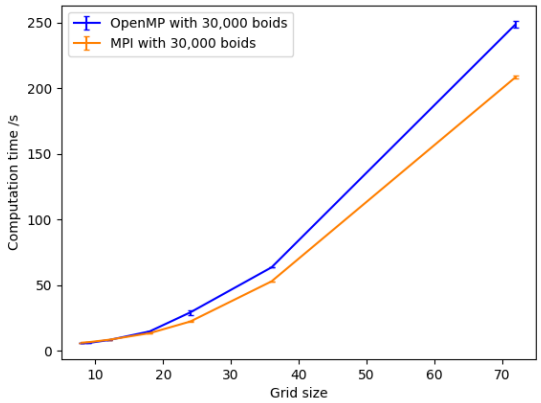


Figure 9: Computation time against grid size using OpenMP and MPI on BC4 with four threads and $30,000$ boids.

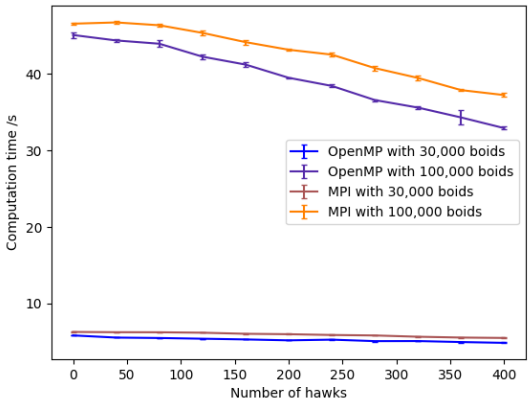


Figure 10: Computation time against number of hawks using OpenMP and MPI on BC4 with four threads and $30,000$ and $100,000$ boids.

4 Discussion

4.1 Scaling with system size

Figure 4 shows how the computation time increases as the number of boids is increased using various numbers of threads with OpenMP. For very small systems, the log plot shows that adding more threads slows the program down since the overhead associated with running the threads is greater than the speed-up gained from running the core loop with more threads. The log graph has a shallow initial gradient because the time taken is dominated by the serial code so adding more boids barely increases the total time. For system sizes greater than about 10^4 boids, using more threads lowers the computation time.

Figure 5 shows a similar graph for MPI which has the same shape as the OpenMP one at high numbers of boids however, the times are slightly slower. This is because as the system size and the number of threads increases, MPI's communication becomes more arduous as more data needs to be transferred to more places. However, OpenMP does not suffer from this problem as the memory is all shared and all the cores are on the same node - although delays arise when multiple threads request access to the same memory. The MPI data also shows that for small systems with around 10^3 boids, some speed is gained by using multiple threads but the optimal value is around 14 threads instead of one of the extremes. The gathering communication speed could be improved by extending the arrays sent so they are the same length, allowing the use of $Gather$ instead of $Gatherv$. The added elements could then be removed when the data is organised by the manager rank. $Gather$ is faster than $Gatherv$ as it does not require the metadata to organise combining the variable length arrays. Additionally, the extra elements sent are few compared to the length of the whole array as the allocation method gives each rank a similar quantity of work.

Using 28 threads, the gradients of the log plots were 1.50 and 1.42 for OpenMP and MPI respectively. The expected scaling with system size is quadratic as each boid adds another boid to calculate the motion of as well as a boid for other boids to potentially interact with. This should give a gradient of two on the log plot. For 28 threads, the measured gradients are lower because as boids are added, the program also becomes more efficient as the parallel parts are given more work to do which reduces the proportion of time spent running serial code, initialising the $prange$ in OpenMP and communicating in MPI. With a single thread, the gradients were 1.85 and 1.68 for OpenMP and MPI respectively which is closer to the quadratic scaling because a single thread does not benefit from the parallelisation becoming more efficient with system size. MPI's gradients are lower than the corresponding OpenMP ones because MPI's additional serial code (including communication, organising the ranks and recording the order boids are calculated in) increases the logarithm of the computation time more for the smaller systems, making its gradient shallower.

The standard deviation of the points is small and is the result of background processes and the random initial conditions.

4.2 Scaling with threads

Figure 6a shows that, for a small system of $30,000$ boids using OpenMP and MPI, the MacBook is generally faster than BC4. However after four threads on the MacBook, no more speed-up is seen with OpenMP as efficiency cores are used and MPI worsens due to the number of ranks staying at four causing the threads to time swap. In comparison, BC4 continues to improve with more threads and is aided by the high bandwidth and low latency communication between the cores.

Larger systems of $100,000$ and $250,000$ boids were computed on BC4 and are shown in Figure 6b. With $250,000$ boids on one thread, OpenMP is significantly slower than MPI which, as the two programs share the majority of their code, must be caused by the $prange$ and the management of the shared memory, whereas MPI communication is instant as the manager rank does not communicate with itself.

Figures 6c and 6d show the speed-up and efficiency of the data from Figures 6a and 6b. The speed-up using 28 threads greatly increases as the system size increases and reaches 17.02 and 12.32 for $250,000$ boids calculated using OpenMP and MPI respectively. The difference in these values is primarily caused by OpenMP's larger computation time with a single thread. However, when the threads are increased, MPI loses its lead due to communicating and organising the data calculated by each rank in serial after every time step. Eventually, OpenMP is slightly faster because accessing shared memory is quicker than explicit communication.

The speed-up increases the fastest at low thread numbers but for smaller systems, it quickly starts to asymptote; for example, the speed-up with 28 threads for $30,000$ boids is around four. The efficiency also reflects this - for the small systems, it rapidly decreases and ends at less than 0.2 for 28 threads, whilst for large systems, it only decays slowly.

4.3 Tuning

Figure 7 shows that changing the scheduling of OpenMP's $prange$ does not make a significant difference to the timing for a range of system sizes and thread numbers. Dynamic scheduling was marginally faster for the larger system sizes and was used for all the other figures using OpenMP. It is unsurprising that the difference is negligible as the $prange$ iterates through over $1,000$ grid cells making it unlikely for one thread to have much more work than another. Additionally, most of the grid cells have a comparable number of boids.

Changing the allocation method for MPI resulted in very little variation of the computation times for a range of system sizes and thread numbers as shown in Figure 8. Fortunately, this means that methods which guarantee that the allocation is balanced can be used without sacrificing computation time. Due to the blocking communication, uneven allocation would lead to one thread delaying the whole program and would become more impactful as the system size increased. The estimate for the work given to each cell assumed that the time scaled linearly with the number of boids in the cell. However, the time scaling should be quadratic and was observed using the gradients of the log plots (Figures 4b and 5b) to have a power of 1.42 for large systems using MPI with 28 threads. Therefore, summing the boids in each cell raised to one of these powers could give a better allocation. Using a combination of the cyclic and balanced allocation methods could be the fastest method while preventing uneven allocation.

Figure 9 shows how much time the grid saves as the computation time rapidly increases as the grid cells are made larger even for a small system size. However, the grid size doesn't affect the scaling with system size since as the density of boids increases, the number of boids in the three-by-three square of grid cells surrounding each cell increases with it. For very small systems, using the grid to filter the interactions is slower than checking all the interactions because using more grid cells results in initialising more loops which is significant if the calculations are fast.

Hawks affect the computation time in several ways. They increase it as their motion has to be calculated and they can drive boids to a higher density which increases the interactions. On the other hand, they decrease the computation time by eating boids causing them to be skipped in future time

steps and to no longer interact with other boids. Additionally, boids fleeing from hawks are faster to compute as they stop flocking which is more costly. Figure 9 shows the decrease in computation time dominates especially for large system sizes. The most important factor is that fewer boids require flocking calculations. However, if even more hawks were added or if the simulation used more time steps, the number of eaten boids would be more significant relative to the high initial number of boids which would increase the weight of this factor.

5 Conclusion

OpenMP and MPI were successfully used to speed up the boids simulation with maximum speed-ups of 17.02 and 12.32 respectively on BC4 with 28 threads and $250,000$ boids. An even larger system is likely to have better speed-ups. However for small systems, poor speed-ups were seen with 28 threads and the efficiency of each core was very low. The computation time was found to scale with the system size less than quadratically due to other factors like the parallelisation becoming more efficient with larger systems. OpenMP was generally faster than MPI due the memory being shared between threads whilst MPI required communication and organisation of the output from each rank. For tuning the parameters of the system, the grid size should be the same as the vision radius as increasing it only increases the computation time. Changing OpenMP's scheduling and MPI's allocation method was found to not affect the computation time significantly.

This program could be improved by changing from using a grid to using adaptive mesh refinement [6] to reshape a mesh in response to changes in the boid density. For dense regions, this would make the filtered search area closer to the vision area of the boid and for sparse regions, larger cells could be used to decrease the number of cells iterated through. By making some assumptions such as assuming a maximum boid density, the program could be improved to scale linearly with the number of boids [7]. The vision angle of the boids could be removed as although it's more realistic, it prevents the interactions from being symmetrical. For more accurate data, the simulations could have been repeated more and evolved for longer as many of the 100 steps calculated were spent on forming flocks. The -O3 flag could be used when compiling for an optimised result. The speed-ups using different parallelisation modules such as Numba could be tested, as well as running the program on a GPU using CUDA. Finally, the system could be extended to three dimensions.

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] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[3] Marc Snir. *MPI–the Complete Reference: the MPI core*, volume 1. MIT press, 1998.

[4] Derek L Eager, John Zahorjan, and Edward D Lazowska. Speedup versus efficiency in parallel systems. *IEEE transactions on computers*, 38(3):408–423, 1989.

[5] Xian-He Sun and Yong Chen. Reevaluating amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010.

[6] Tomasz Plewa, Timur Jaan Linde, Vincent Gregory Weirs, et al. Adaptive mesh refinement, theory and applications. 2005.

[7] Norihiro Maruyama, Daichi Saito, Yasuhiro Hashimoto, and Takashi Ikegami. Dynamic organization of flocking behaviors in a large-scale boids model. *Journal of Computational Social Science*, 2:77–84, 2019.

This work was carried out using the computational facilities of the Advanced Computing Research Centre, University of Bristol - http://www.bristol.ac.uk/acrc/.