An Investigation of HPC Techniques Using Boids

Neil Kalnins-Goode

Abstract

The Boids artificial life program was simulated with parallel computing techniques. OpenMP and MPI were used for testing shared and distributed memory systems respectively. The effects of increasing the simulation size and the available threads were tested.

I. Introduction

Modern computers are all now capable of parallel processing. This lets programs use more than one processing core at once to greatly improve the speed however parallel programs are much harder to program efficiently than serial programs. To write efficient code it is important to understand the architecture of modern processors. The von Neumann architecture used by all modern PCs has a CPU with the control unit and the processing unit, and memory connected by a bus that stores data and instructions. Within modern CPUs there is an even faster set of memory called cache. Supercomputers have expanded this to distributed systems, where conventionally separate systems have been connected using extremely high speed interconnects to work on problems together.

Parallel processing can be used to accelerate complex problems. This is a report of a program called Boids that was used to investigate these techniques.

i. Parallel processing

When programming for parallel computers, it is important to identify what sections of a program can be parallelised and which must be serial. Dependencies on other calculations limit how much speed up can be. Amdahl's law states that the maximum speedup that can be obtained by parallelisation is

$$S = \frac{1}{1 - p + \frac{p}{s}} \tag{1}$$

where p is the proportion of the program that can be parallelised and s speedup of the parallel section of the program.

OpenMP is an API for shared memory parallel processing. Allowing parallel tasks to access the same memory can be very fast as data does not have to be transferred between pools of memory however care must be taken to ensure there are no data race conditions. MPI is an API for distributed memory parallel processing. Each thread is created with its own pool of memory and is treated separately. This allows programs to be run on distributed systems where you have multiple computers connected with high speed interconnects like supercomputers. When using distributed memory the data from each thread must be communicated with other threads, slowing down execution.

ii. Boids

Boids is an artificial life program from the late 80s that simulate birds' flocking behaviour [1]. Boids is similar to other artificial life programs [2], it is made to show emergent behaviour of individual agents interacting with simple rules.

A simple Boids program has 3 rules. Separation means Boids will avoid colliding with

each other. Alignment means that Boids will try and match the average heading of the surrounding Boids. Cohesion means that Boids will seek towards the centre of mass of their local Boids. The Boids will only be aware of other Boids in their local surroundings.

II. Methods

The program starts defining the initial conditions of the simulation. The initial positions are set with random Cartesian coordinates within the bounds of the simulation. The initial directions are set as a random angle and converted into a vector of a set magnitude.

Finding the next position and direction of each boid is a serial process. The first step is finding the local boids. A second boid will be defined as local to the first boid if it is within a defined distance and inside the field of view of the first boid. For each boid, every other boid is checked. Using only the local boids the steering contributions from the separation, alignment, and cohesion are calculated.

i. Boids rules

The steering contribution of separation is defined by equation 2 where $\vec{d_s}$ is the separation steering direction, $\vec{p}_n$ denotes the position vector of the current boid and $\vec{p}_{local}$ denotes the position vector of any local boid. This results in the boids repelling each other with magnitude proportional to $\frac{1}{r^3}$

The steering contribution of alignment is defined by equation 3 where $\vec{d_a}$ is the alignment steering direction, $\vec{d_n}$ denotes the direction vector of the current boid, $\vec{d}_{local}$ denotes the direction vector of any local boid, and N is the number of local boids. This will steer the direction of the current boid closer to the average direction of local boids.

The steering contribution of cohesion is defined by equation 4 where $\vec{d_c}$ is the cohesion steering direction. This will steer the boid towards the centre of mass of the local boids.

$$\vec{d_s} = \sum_{local} \frac{\vec{p}_n - \vec{p}_{local}}{|\vec{p}_n - \vec{p}_{local}|^4} \tag{2}$$

$$\vec{d_a} = (\sum_{local} \frac{\vec{d}_{local}}{N}) - \vec{d_n} \tag{3}$$

$$\vec{d_c} = \sum_{local} \frac{\vec{p}_{local} - \vec{p}_n}{N} \tag{4}$$

The next direction of the boid is then calculated by equation 5 where C_s, C_a, and C_c are the coefficients of each contribution. The next position of the boid is calculated by equation 6 where h is the time between each step.

$$\vec{d}_{next} = \vec{d_n} + C_s\vec{d_s} + C_a\vec{d_a} + C_c\vec{d_c} \tag{5}$$

$$\vec{p}_{next} = h \cdot \vec{d}_{next} \tag{6}$$

ii. Shared memory

As discussed, progressing each boid is a serial operation but it does not depend on any other operation. This means that using OpenMP on this program only required making a parallel for loop. The output of the loop was set to a new data structure in memory to avoid data race conditions.

This method is expected to be very fast, as there is no synchronisation step between memory pools.

iii. Distributed memory

The solution for MPI is a little more complex. Each MPI node has its own pool of memory so the data needed for each node needs to be shared from other nodes as a message. Sharing messages between nodes requires the sender and receiver to both actively send and receive the message respectively. If one node is ready before the other, it must wait. Generally these systems will be set up in a master and worker scenario where there is one master node that sets up the other nodes and manages data from the other nodes.

For this program, each node must contain the entire data-set of the previous positions

and directions for boids. Transferring this data using the individual send and receive functions would be a huge undertaking. MPI has built-in functions that make this much easier. The functions Bcast and Allgather are built to transfer larger data-sets to multiple nodes. Bcast will send a whole data-set from one root node to all other nodes. This will be used to send the initial state of the boids to all the nodes. Once the data has been sent, the nodes will start their work. Each node is given a list of boids that it must progress the state of. Once this is finished the nodes then begin the Allgather function. This function will gather all the data in the smaller buffers in each node, gather them all into one structure, and then send that structure to every node. This allows the program to function without a master node between the initial setup of the system and the output of final data.

The downside to this method is that there is no way to do load balancing as the function will only allow fixed size buffers. This is not a big issue for this program as calculating the next step each boid should take roughly the same amount of time so load balancing should not be needed anyway.

iv. Time complexity

The algorithm of each boid will start by searching through $n-1$ boids. This will be done once for each boid, n times. From this it can be deduced that the time complexity of the program should be $O(n^2)$. This is assuming that this first step takes much longer than the other steps to complete when n is large.

v. Visualisation

The results of the program were visualised using matplotlib in python. The data was transferred by saving every position of every boid in every frame to a text file. This was then loaded into python and parsed into an array that could be animated. This was a crude method but was sufficient for the needs of this experiment as there was not any data collection to be done from this. The visualisation was done as an eye test to make sure the program is working as intended.

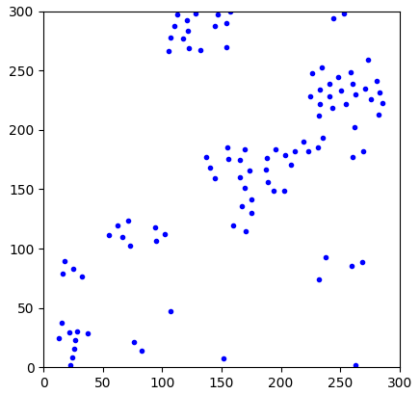

Figure 1: *A screenshot of the matplotlib animation output from the python program.*

vi. Hardware

The testing for the OpenMP and MPI sections was completed on BlueCrystal4. BlueCrystal4 is a supercomputer at University of Bristol [3]. This system has 525 compute nodes, each of which has two 2.4 GHz 14 core Intel E5-2680 v4 CPUs and 128GiB of RAM.

The testing for the time complexity section was completed on a desktop with a 4.4 GHz Ryzen 7 5800x 8-core 16-thread CPU with 32GiB ram [4].

III. Results

i. Boids

Figure 1 shows a screenshot of the animation midway through a small simulation. The boids will wrap around when crossing over the boundary so that they stay within the area. In the screenshot there is definite flocking behaviour however the intricacies of the behaviour greatly rely on the values of the chosen coefficients for separation, alignment, and cohesion.

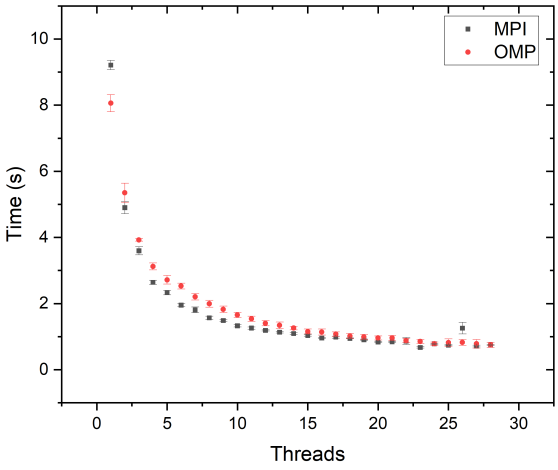

Figure 2: *A graph showing the time taken to compute 3000 frames of Boids with 800 boids compared to the number of threads used. This was done on BlueCrystal4.*

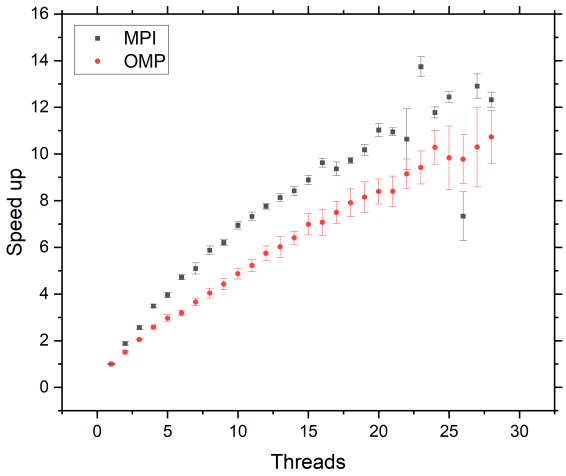

Figure 3: *A graph showing the speed up compared to 1 thread when computing 3000 frames of Boids with 800 boids. This was done on BlueCrystal4.*

ii. OpenMP

All timings were done using a simple timer script that used std::chrono [5].

The results from the OpenMP tests shown in figures 2 and 3 show that there are huge gains that can be made from parallelising this program. For higher thread counts, the speed up gained from adding more threads tapers off.

iii. MPI

The results from MPI testing shown in figures 2, 3, 4, and 5 show that there are similar gains to be made from using MPI as there are in OpenMP. The gains in speed stop once the simulation uses more than 28 threads.

The data from the MPI simulations appears to be very noisy with big discrepancies between runs with similar thread counts, especially when using more than 28 threads.

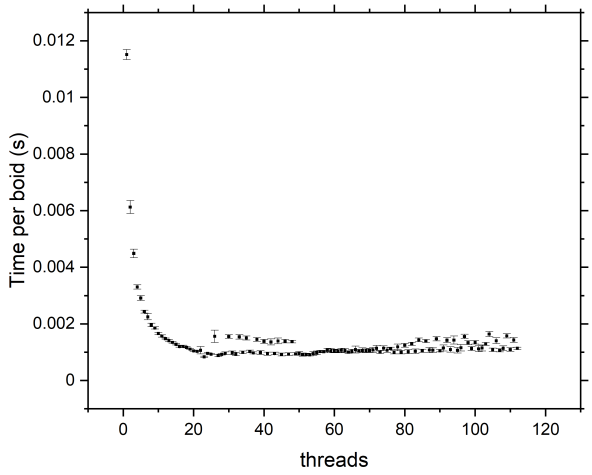

Figure 4: *A graph showing the time taken to compute 3000 frames of Boids with 800 boids compared to the number of threads used using MPI. This was done on BlueCrystal4*

iv. Time complexity

The time complexity results are shown in figure 6. A 2nd order polynomial was fitted to each set of data to test the hypothesis of the algorithm's time complexity. The r-squared

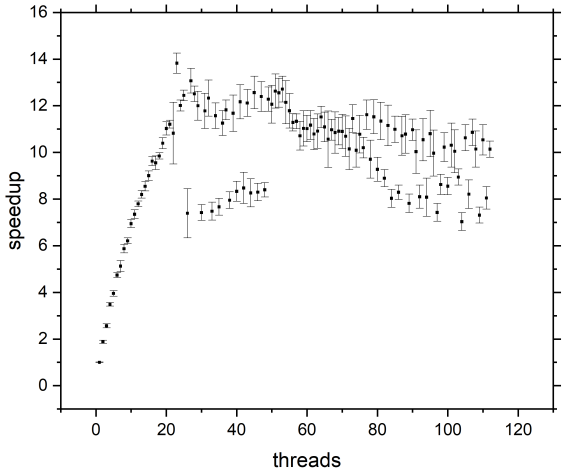

Figure 5: *A graph showing the speed up compared to 1 thread when computing 3000 frames of Boids with 800 boids using MPI. This was done on BlueCrystal4.*

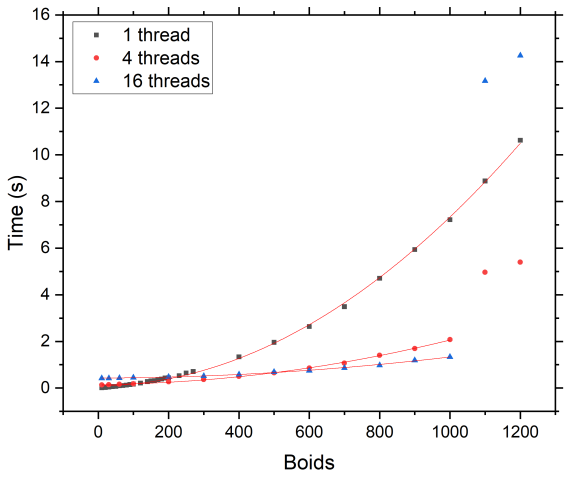

Figure 6: *A graph demonstrating the time complexity of Boids using a simulation of 3000 frames.*

value of the 1 thread data set was 0.99948, for 4 threads it was 0.99905 and for 16 threads it was 0.99671.

There was an interesting result that appeared in the 4 and 16 thread data where over 1000 boids the performance slowed down heavily. The slowdown of the 16 thread data was much more severe than the 4 thread data.

IV. Discussion

i. Visualisation

The visualisation used for this Boids program was very crude as not much more was necessary for this experiment however this is not a good way for Boids to be shown. Boids is a fairly light program on modern CPUs, one BlueCrystal 4 core can process 3000 frames using 800 boids in under 10 seconds. This works out to about 3 ms per frame, much less than the 16ms needed for real-time 60Hz rendering. It makes much more sense to program Boids in a game engine such as Unity as it would allow the system to be visualised much easier. The game engine would also make it much easier to add advanced features such as object avoidance or object seeking, as shown in the original Boids paper [2] and some more modern examples [6]. It is actually very common to find Boids in games, an oft mentioned example being 1998's Half-Life. These Boids simulations are usually using far less than the number of boids processed in this experiment so they will be very cheap to process.

ii. OpenMP vs MPI

Results from the testing showed that the speed up as the number of threads is increased tapers off, although quite slowly in this example. This suggests that the data is following Amdahl's law as shown in equation 1. It is hard to determine any concrete data from this however, as the OpenMP program can only use 28 cores on BlueCrystal4 and when the MPI program reaches over 28 cores the program stops speeding up because of the architecture of the system.

The results from the MPI testing showed that for simulations using more than 28 cores, there is no longer any more speed up. One node of BlueCrystal 4 only has 28 cores. Simulations using more than 28 threads must use cores from different nodes. The communications between these threads will have much higher latency and lower bandwidth than threads on the same node and will cause a lot of slowdown.

When first beginning this experiment, the OpenMP solution was expected to be much faster than the MPI solution as the implimentation is much more complex. This has proved not to be the case, Figure 2 shows that most of the time the MPI solution is actually much faster than the OpenMP solution. The reasons for this are not clear at all however it is a good sign that this solution and MPI as a whole are very fast.

These results generally showed that this program is not very well suited to running on supercomputers like BlueCrystal4. The time taken for each frame is so small that the effects of slower bandwidth and latency storage for the data has a much larger impact on the performance than using more cores. These results may change for much larger sets of boids however running these simulations on BlueCrystal can be difficult as they take a long time and can sit in the job queue indefinitely.

iii. Time complexity

The results from the time complexity tests support the hypothesis that the time complexity of the algorithm is $O(n^2)$ because all three 2nd order polynomials fit the data very well, with r-squared values very close to 1.

The interesting result is that there was huge slowdown for the 4 and 16 threaded programs when reaching over 1000 Boids shown in figure 6. The amount of data stored for each boid is 32 bytes as it is made up of four doubles. This means that for a simulation of 1000 boids, there is 32 KB of data or 31.25 KiB. This test was performed on the 5800x system. The L1 cache of the 5800x is made up of a 32KiB data store and 32KiB instruction store [4]. It is very likely that the slowdown is caused by swapping data in and out of the cache from RAM. This could also be why the 16 threaded run had a much larger hit as it will need to swap data in and out of 16 caches. The 1 thread run seems unaffected by this. This could be because the data is stored in a larger higher level cache when only using 1 thread.

V. conclusions

Increasing the core count of the simulation can bring huge speed ups however the scaling is not perfect and there are lots of factors that can affect performance. The Boids program is fairly lightweight, with each iteration being performed in around $3ms$. The latency of the data transfer is much more impactful to the overall performance than the number of cores used. Boids in this form is not particularly well-suited to HPC systems like BlueCrystal and would be presented much better inside a game engine.

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*. 1987, pp. 25–34.

[2] Martin Gardner. "Mathematical games". In: *Scientific american* 222.6 (1970), pp. 132–140.

[3] *BlueCrystal Phase 4 - ACRC, University of Bristol*. URL: https://www.acrc.bris.ac.uk/acrc/phase4.htm (visited on 02/07/2022).

[4] *Ryzen 7 5800X - AMD - WikiChip*. URL: https://en.wikichip.org/wiki/amd/ryzen_7/5800x (visited on 02/06/2022).

[5] *C++ Timer using std::chrono · GitHub*. URL: https://gist.github.com/mcleary/b0bf4fa88830ff7c882d (visited on 02/07/2022).

[6] Sebastian Lague. *Boids*. original-date: 2019-08-26T12:18:17Z. Feb. 2022. URL: https://github.com/SebLague/Boids (visited on 02/07/2022).