Parallelization of Boids Algorithm

Cristiano Narducci
E-mail address
cristiano.narducci@edu.unifi.it

Abstract

The work presented here introduces a C++ implementation of Boids algorithm, or "storni" in Italian, and how to emulate their behavior when faced with other boids. This work was carried out using the OpenMP library to leverage parallelization technologies and the resulting advantages. We used two different approach for memorize the Boids information: Array-Of-Structure and Structure-Of-Array and compared themself to see which structure perform better for the model. In addition, we've used a graphical visualization library for visualizing the flocking object behavior for a better comprehension of how this algorithm works.

1. Introduction

1.1. Boids Algorithm

The Boids (Bird-oid object) program is an artificial program life (**ALife**) program, developed by Craig Reynolds in 1986, whose objective is to emulate the behavior of small groups of birds [2]. The Reynolds program is an example of the implementation of a general concept. Many other scientists implemented their own version of the program,such as Ichiro Aoki working in an acquatic scenario [1].

1.2. The model

The most interesting part of this model is that his general behavior is conditionated by the iterations of all small agents with themselves, according to a set of rules. These rules are as follows:

- **Separation**: boids move away from other boids that are too close.
- **Cohesion**: boids attempt to match the velocity of their neighbors.
- **Alignment**: boids moves towards the center of mass of their neighbors.

In addition, more rules can be added for a complexity study, such as goal seeking or obstacles avoidance. In the work provided, it was added a sort of boundaries rule, representing the display size, in the way that the boids are forced to change theirs position for remain inside the display.

1.3. OpenMP

OpenMP is a multi-platform API for multiprocessing programming in C/C++ and Fortran. Its composed by a series of compiler directives and environmental variables that alter the runtime behaviour. For further information, consider visit the official page https://www.openmp.org.

1.4. System Specification

The program execution was done with an Apple MacBook air with the following specification:

- CPU: Apple M1 8 core processor
- RAM: 8 GB

for the delevoping environment we used Clion 2025.2.4 and OpenMP ver. 5.1.

1.5. Structure Configuration

For the purpose of the problem, we used two types of structure for representing the boids: **Array of Structure** and **Structure of Array**. They

represent two different ways to structure some data. It was done because It is important, when dealing with CPU cycle and optimization, to get some cache friendly data. We compared them to see which of this method of data memorization performs better.

2. Algorithm Description

2.1. Boids Behavior

As said in the introduction, the algorithm produces a realistic simulation of the flocking behavior. Like the real world, every flying object doesn't know every other flock. Instead, **it only consider the boids that are in his visual range**.

2.1.1 Separation

A boid will move out from other boids if they will be in his protected range, so as not to collide with them. This behavior is called, as said previously, **separation**.

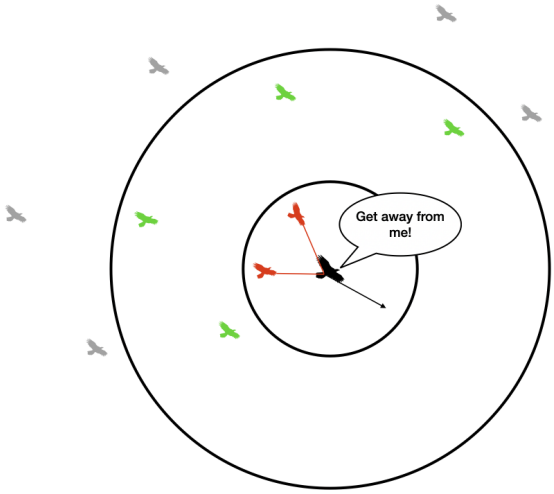

Figure 1. Separation

2.1.2 Alignment

The **alignment** property said that every boid will match the velocity of the others inside his visible range. The algorithm will calculate the average velocity within the range and set up the flock velocity times a numerical factor.

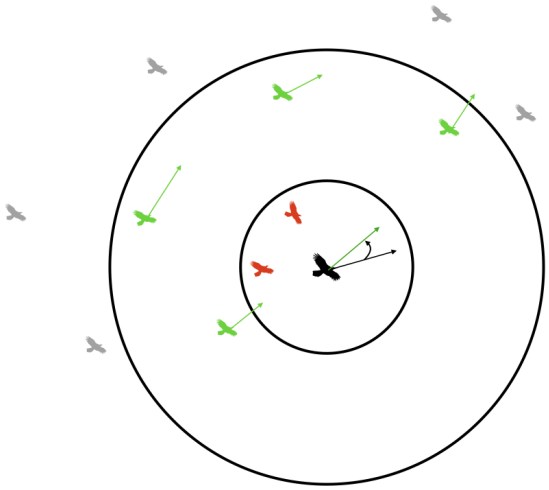

Figure 2. Alignment

2.1.3 Cohesion

Every boid will try to gently move toward the center of the mass of the others in his visible range. This feature is implemented by checking the neighboring boids and scaling the velocity by a matching factor tunable by the programmer.

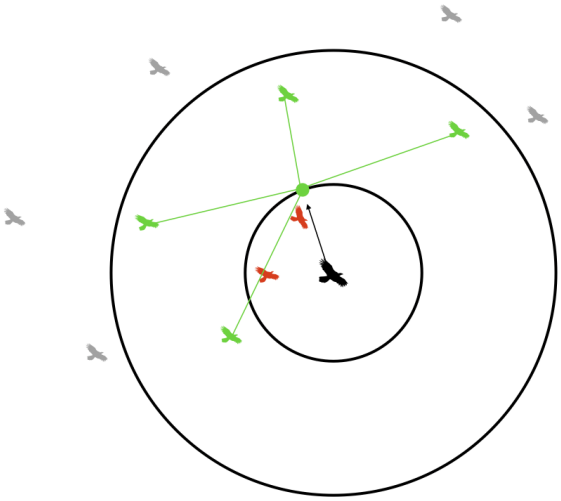

Figure 3. Cohesion

2.1.4 Screen Edges

For the purpose of the experiment, we implemented a sort of *screen borders* the boids need to be within. They will turn around when approach

the fixed margin by a tunable factor

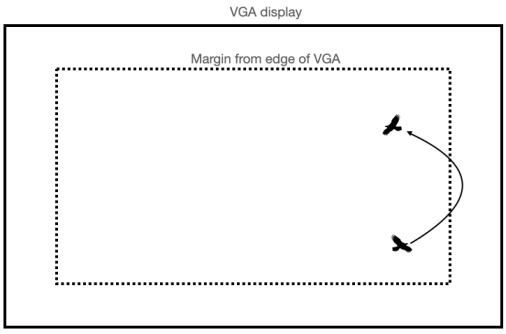

Figure 4. Screen Edges

2.2. Speed Limit

The program constraints the object to move within a range of possible horizontal and vertical velocity. We had set a minimum and maximum value that every boids can have for their speed. we checked those velocity calculating the next value: $v = \sqrt{(v.x)^2 + (v.y)^2}$

3. Implementations Strategy

The main strategy for implements the flocking behavior previously enunciated is encapsulate every property in a single sequential function. when there was need to applying the rules, we call the function for every boid and compared his positions and velocities with others. When a property was violated, the program calculate the new position by the methods wrote in the paper.

3.1. Implementation of Parallelization

For the main purpose of the analysis, we created a parallelized section of the code. A parallelized section is created at runtime, in particular, the procedure follows those steps:

1. A first for cycle assigns a number of boids per thread to calculate the new velocities according to the rules.

2. A second cycle set the new position at each boids

This separation was done because we wanted to eliminate possible race conditions, in particular when assigning the new positions.

3.2. Graphical visualization

To visualize how these rules affect the movements, we used a graphical library named **SFML**. Every boid is represented by a circle shape, and a separated code section updates the shape position in the window. The measurements of computation times of the parallelized section are done in a different section, which excludes the times needed for visualizing the flocks.

4. Execution and Results

For the experiment, we defined the data that needs to be compared:

- **NUM BOIDS**: number of the flying objects
- **NUM STEPS**: number of steps that the boids will take
- **WIDTH**: width of the available area
- **HEIGHT**: height of the available area

After we set the factor that change the positions of the objects:

- **VISUAL RANGE**: radius of visible boids per a single entity
- **PROTECTED RANGE**: radius whereby when a flock is inside, the entity tends to move away
- **CENTERING FACTOR**: Describe how much a they move towards the center of the mass of others
- **AVOID FACTOR**: numerical factor of how fast a boid avoid the others inside is protected range
- **MATCHING FACTOR**: how fast they match the velocity of flocking inside the visible range
- **TURN FACTOR**: how fast the tend to avoid the margin of the display

- **MAX SPEED**: maximum possible velocity
- **MIN SPEED**: minimum possible velocity

4.1. Execution

We started the analysis with a small number of boids (1000 elements) with the same number of steps, and we gradually incremented the first one to see how the different data structures perform.

Elements Number	Time of AoS	Time os Soa
1000	0.605 s	0.444 s
5000	10.702 s	8.919 s
7000	23.59 s	19.30 s
10000	44.385 s	40.45s

Table 1. Execution Results

The results of the sequential version are shown below:

Number of boids	Sequential AOS	Sequential SOA
1000	1.621 s	1.608 s
5000	41.467 s	41.045 s
7000	80.285 s	80.201 s
10000	163.074 s	162.563s

Table 2. Sequential results

4.2. Speedup

As seen in class, the most important thing to notice is the effective advantages that parallelization offers when used effectively. The formula to see that is called **speedup**:

$$Speedup = \frac{T_{seq}}{T_{par}}$$

where T_{seq} is the duration of the sequential version of the code and T_{par} is the time of the parallelized mode. The next table shows the evaluation of the speedup of Boids algorithm:

Number of boids	**AOS Speedup**	**SOA Speedup**
1000	1.632	1.618
5000	3.878	4.557
7000	3.790	11.095
10000	4.044	4.544

Table 3. Speedup Evaluation

4.3. Result Evaluation

As we saw in the last section, the structure of array's type performs better than the other one. Those results happen because in SoA mode **every boids detail is stored in a dedicated vector**, so when dealing with calculate the distance for two flocks is faster then access with respect to access the single instance. If we needed to compute all the coordinates and the velocities instead, AoS should perform better.

References

[1] I. AOKI. A simulation study on the schooling mechanism in fish. *NIPPON SUISAN GAKKAISHI*, 48(8):1081–1088, 1982.

[2] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, page 25–34, New York, NY, USA, 1987. Association for Computing Machinery.