Boids: sequential version vs parallel version

Bruno Martelli
E-mail address
bruno.martelli@stud.unifi.it

Abstract

This report aims to provide an in-depth analysis of the advantages and disadvantages associated with parallelizing the proposed algorithm [1]. The discussion will explore how transitioning from a sequential implementation to a parallel one can influence computational efficiency, with specific focus on metrics such as calculation time and speedup.

1. Introduction

The ”Boids” program, developed by Craig Reynolds in 1986, simulates lifelike flocking behavior through a set of simple rules applied to individual agents known as ”boids” (short for ”birdoid objects”). The main rules are:

1.1. Separation

This rule ensures that each boid steers away when it gets too close to another boid within a defined protected range. The Separation behavior helps maintain proper spacing between boids, preventing overlap and collisions.

1.2. Allignment

This rule causes each boid to adjust its velocity to match the average velocity of nearby boids. By doing so, each boid ensures that it moves in harmony with the rest of the flock, contributing to a cohesive movement

1.3. Cohesion

This rule directs each boid to steer gently toward the center of mass of nearby boids within its visible range. This behavior encourages the boids to move closer together, promoting group unity and ensuring the flock remains cohesive.

1.4. Other rules

After the main three rules, additional rules are implemented as described in [1], such as the screen edge rule, which directs boids to steer in order to stay within the screen boundaries, and the speed limit rule, which prevents boids from either remaining stationary or moving too fast.

2. Code Implementation

For the previously described program, I decided to implement it in a few different ways, which I will detail in the following subsections.

2.1. Naive Implementation

The initial implementation is naive and based on an Array of Structures (AoS) approach. In this setup, we have an array of Boids, where each Boid has its own velocity and position.

Listing 1. Boids AoS

```
struct Boid
{
    Boid(const sf::CircleShape& body, const
        sf::Vector2f& velocity = sf::Vector2f
        ((rand()%(2*(int)maxspeed))-maxspeed,
         (rand()%(2*(int)maxspeed))-maxspeed)
         ):
        body(body), velocity(velocity) {}

    sf::CircleShape body; //contains position
    sf::Vector2f velocity;

};
```

To make the update position phase as simple as possible, we implemented three functions, each one applying a main rule:

Listing 2. sequential update

```
    void Flock::update(){
      for (auto& boid : all_boids){
        boid.velocity += cohesion(boid)+
              alignment(boid)+separation(boid);
        float spd =utility::length(boid.
            velocity);
        if (spd>maxspeed) {
          boid.velocity = boid.velocity*
                  maxspeed/spd;
        }
        if(spd<minspeed) {
          boid.velocity = boid.velocity*
                  minspeed/spd;
        }
        sf::Vector2f p =boid.body.getPosition()
            ;
        checkBoundaries(boid);
        boid.body.setPosition(p+boid.velocity);
      }
  };
```

2.2. SoA implementation

The SoA (Structure of Arrays) paradigm is one of the main ideas for optimization. Applied to our case, it led us to substitute the original `boid` structure with a new structure, ,`Boid_SoA`, which contains separate arrays for the positions, velocities, and sprites of all the boids.

Listing 3. boids SoA

```
  struct Boid_SoA{
       std::vector<float> positions;
       std::vector<float> velocities;
       std::vector<sf::CircleShape> shapes;
       };
```

2.3. Fast implementation

The "Fast Implementation" is the name I gave to an optimization of the update process, which involves aggregating all the main rules into a single for loop. This approach achieves a performance improvement due to the reduction in the number of for loops from three to one.

Listing 4. Sequental Update with SoA and Fast

```
void Flock::updateFastSequential() {
    for (int i  = 0; i < nBoids; i++){
      float px = all_boids.positions[i*2];
      float py = all_boids.positions[i*2+1];
      float vx = all_boids.velocities[i*2];
      float vy = all_boids.velocities[i*2+1];
      float temp_vx = 0;
      float temp_vy = 0;
      float temp_px = 0;
      float temp_py = 0;
      float temp_vx_s = 0;
      float temp_vy_s = 0;
      int neighbors = 0;
      int neighbors_s = 0;
      for(int k =0 ; k<nBoids; k++) {
        if(i != k) {
          float distance = utility::distance(px,
              py, all_boids.positions[k*2],
              all_boids.positions[k*2+1]);
          if (distance < visualRange) {
            temp_vx += all_boids.velocities[k*2];
            temp_vy += all_boids.velocities[k
                *2+1];
            temp_px += all_boids.positions[k*2];
            temp_py += all_boids.positions[k
                *2+1];
            neighbors++;
            if(distance < protectedRange) {
              temp_vx_s +=  px - all_boids.
                  positions[k*2];
              temp_vy_s +=  py - all_boids.
                  positions[k*2+1];
              neighbors_s++;
            } } } }
      if (neighbors != 0){
        temp_vx /= neighbors;
        temp_vy /= neighbors;
        temp_vx = (temp_vx - vx )*matchingFactor;
        temp_vy = (temp_vy - vy )*matchingFactor;
        temp_px /= neighbors;
        temp_py /= neighbors;
        temp_px = (temp_px- px)*centeringFactor;
        temp_py = (temp_py- py)*centeringFactor;
      }
      if (neighbors_s != 0){
        temp_vx_s /= neighbors_s;
        temp_vy_s /= neighbors_s;
        temp_vx_s *= avoidFactor;
        temp_vy_s *= avoidFactor;
      }
      all_boids.velocities[i*2] += temp_vx +
          temp_px + temp_vx_s;
      all_boids.velocities[i*2+1] += temp_vy +
          temp_py + temp_vy_s;
      float spd =utility::length(all_boids.
          velocities[i*2],all_boids.velocities[i
          *2+1]);
      if (spd>maxspeed) {
        all_boids.velocities[i*2] = all_boids.
            velocities[i*2]*maxspeed/spd;
        all_boids.velocities[i*2+1] = all_boids.
            velocities[i*2+1]*maxspeed/spd;
              }
      if(spd<minspeed) {
        all_boids.velocities[i*2] = all_boids.
            velocities[i*2]*minspeed/spd;
        all_boids.velocities[i*2+1] = all_boids.
            velocities[i*2+1]*minspeed/spd;
              }
      checkBoundaries( all_boids,  i);
      all_boids.positions[i*2]= px + all_boids.
          velocities[i*2];
```

```
    all_boids.positions[i*2+1]= py + all_boids.
        velocities[i*2+1];
          }}
```

2.4. Parallelizzation

The parallelization is implemented using the OpenMP library and the `#pragma omp for` directive. This method assigns each thread a subset of boids, which are updated with their new positions.

Listing 5. parallel update

```
void Flock::updateFast() {
#pragma omp parallel for schedule(static)
    num_threads(Flock::numthreads) default(none)
    shared(all_boids)
    for (int i  = 0; i < nBoids; i++){
        // same as sequental fasat update
```

2.4.1 Dependencies

Analyzing the sequential version and referring to the explanation of the algorithm in [1], a sequential dependency between the boids becomes evident. For instance, the behavior of the fifth boid depends on the positions and velocities updated for the preceding boids. From the tests we conducted, we observed that removing this dependency causes the boids to form more compact groups, showing greater cohesion. However, considering natural behavior, it is reasonable to assume that some boids move before others in an apparently random manner.

Following this intuition, we decided to keep the position updates within the parallelized loop. This approach, in the parallel version, resulted in a mixed behavior that aligns realistically with the movement of flocks.

3. Results

In this section, we will present a series of experiments conducted on the discussed code, changing the number of threads and the number of boids.

3.1. Setup

The tests were conducted on a PC equipped with a Ryzen 7 3800X 8-core processor operating on Windows 11 Pro 24H2.

3.2. Description of the experiments

The experiments were conducted using 10, 100, 250, and 500 boids. For each case, the average time taken to update positions was measured, and the speedup was calculated by varying the number of threads.

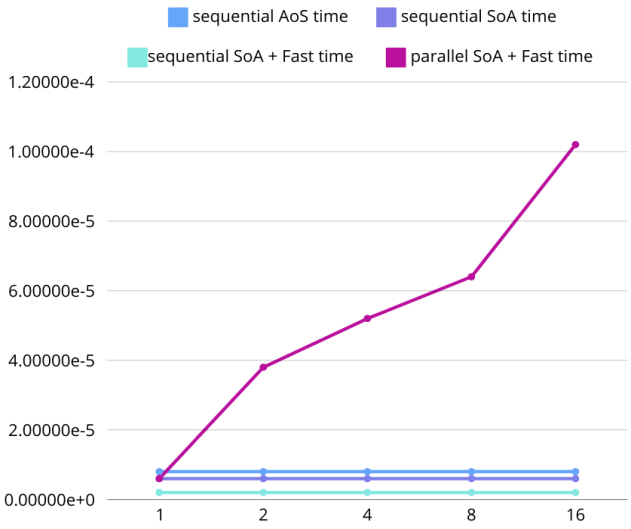

Figure 1. Execution time for 10 boids

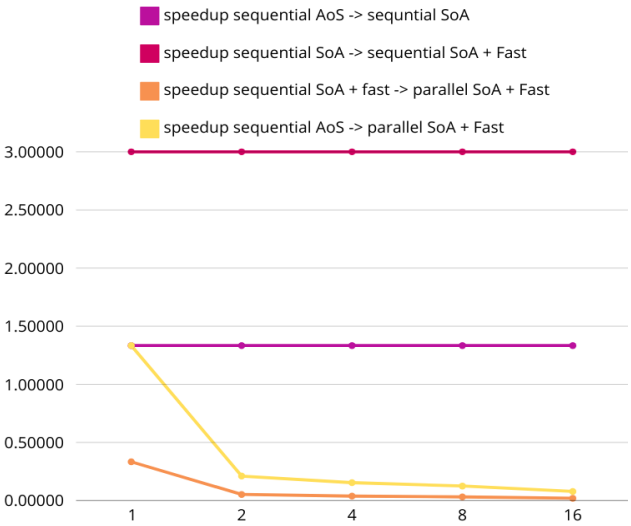

Figure 2. Execution time for 10 boids

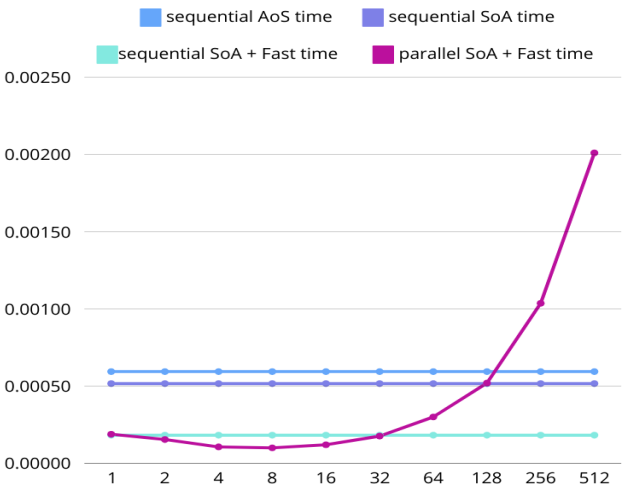

Figure 3. Execution time for 100 boids

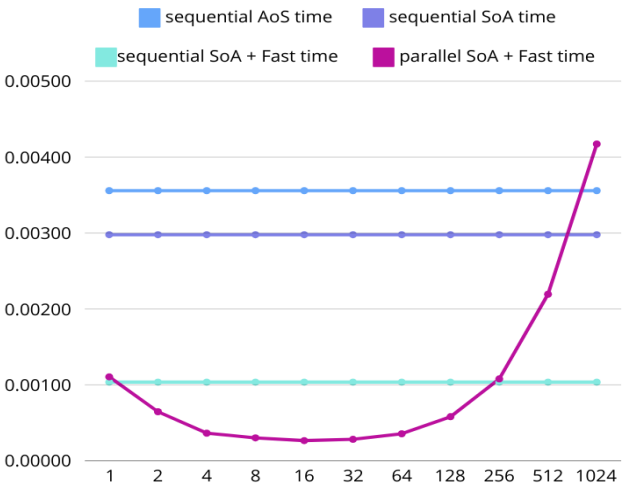

Figure 4. Execution time for 250 boids

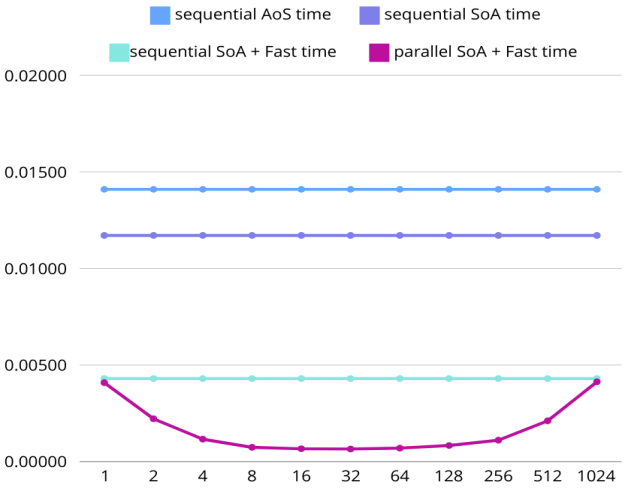

Figure 5. Execution time for 500 boids

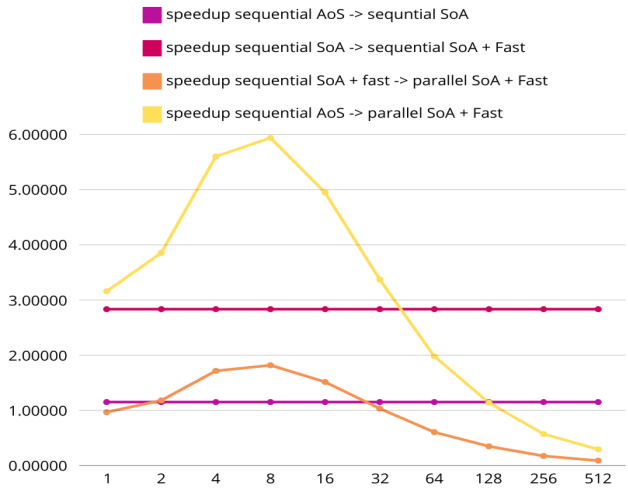

Figure 6. Execution time for 100 boids

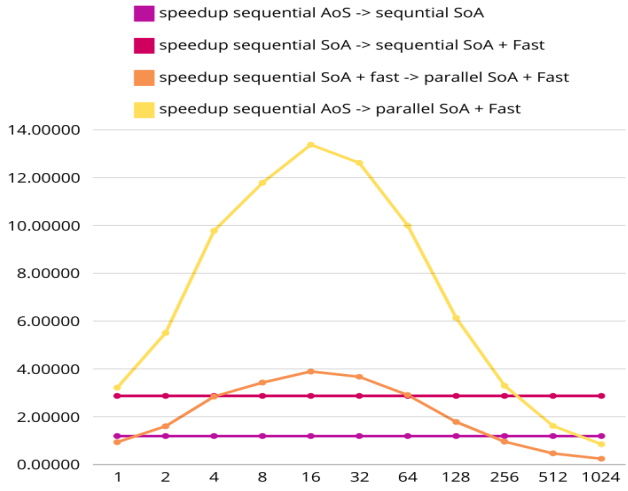

Figure 7. Execution time for 250 boids

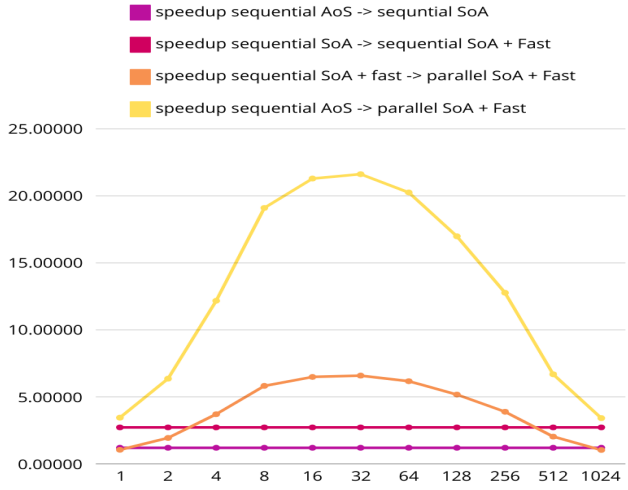

Figure 8. Execution time for 500 boids

3.3. Final Discussion

The analysis of the cases with 100 and 10 boids led to limiting the data representation to enable a clearer understanding of the results. It is evident that the Fast + AoS version is, on average, three times faster than the AoS version alone, consistent with the reduction in the number of for loops from three to one.

In the specific case of 10 boids, an interesting aspect emerges: parallelization does not provide any benefit due to the overhead associated with thread management, which outweighs the advantages of parallelization itself. This also explains the slight slowdown observed in tests conducted with a single thread compared to the fully sequential version.

Another significant result is the observed speedup, ranging between 1.2 and 1.3, when shifting from AoS to SoA. This confirms the importance of optimizations related to memory structure and data management. It is also noted that the optimal number of threads increases with the number of boids, although the differences between various thread configurations tend to become progressively less significant.

Finally, the parallel version demonstrated a speedup of 6.5 compared to the corresponding sequential version and over 20 compared to the sequential naive version. These results clearly showcase the effectiveness of the applied optimizations in improving overall performance.

References

[1] V. H. Adams. Boids algorithm - augmented for distributed consensus. `https://vanhunteradams.com/Pico/Animal_Movement/Boids-algorithm.html`, 2008. [Online; accessed 20-11-2024].