Comparison between sequential and parallel programming on a flocking simulation with OpenMP

Riccardo Fantechi
E-mail address
riccardo.fantechi1@edu.unifi.it

Abstract

In this work, we simulated the behavior of boid flocks. The main goal was to create a visually realistic and efficient simulation using parallelization techniques via OpenMP. Visualization was achieved through SFML, whit a fare rate cap for stable and consistent simulation. To address potential dependencies between boids during updates, we implemented two approaches: one directly modifying boids based on their neighbors and another using a temporary structure to store intermediate results, avoiding implicit dependencies. The introduction of parallelism achieved significant speedup over the sequential version, enabling faster updates and better scalability for large flocks.

Future Distribution Permission

1. Introduction

We will show the development and the implementation of a simulation model inspired by flocking behavior, similar to nature. The model is focused on **Boids** (bird-oid object), which are simple objects that move together based on three basic rules:

- **Cohesion** staying close to others
- **Separation** avoiding collisions
- **Alignment** moving in the same direction

The idea and algorithms were first introduced by Craig Reynolds in 1987 [1], providing a simple yet effective way to model complex collective behavior. We developed both sequential and parallel versions of the simulation. The sequential version serves as a baseline to understand the model, while the parallel version leverages OpenMP to improve performance and handle larger flocks. During the implementation, we considered two strategies to parallelize the updates: one directly modifying boids based on their neighbors and another using a temporary structure to ensure independence between updates. This report compares the two implementations in terms of design and performance, analyzing how parallelization achieves significant speedup while maintaining realistic and stable flocking behavior.

1.1. Setup

The development was conducted on a MacBook Air M1 (2020), with an Apple M1 chip, 8 GB of RAM and running on macOS Sonoma 14.2. To parallelize the code we used the OpenMP library in the Clion editor.

1.2. SFML Library

In this project, we used SFML (Simple and Fast Multimedia Library) to visualize the movement of the boids. SFML is a simple and easy library for creating multimedia applications. It allowed us to display the boids as shapes on the screen, update their positions, and create a smooth simulation. As can be seen from the (Figure 1), a VGA boundary is also shown, it is used to indicate the area beyond which the boids tend to return towards the center of the screen.

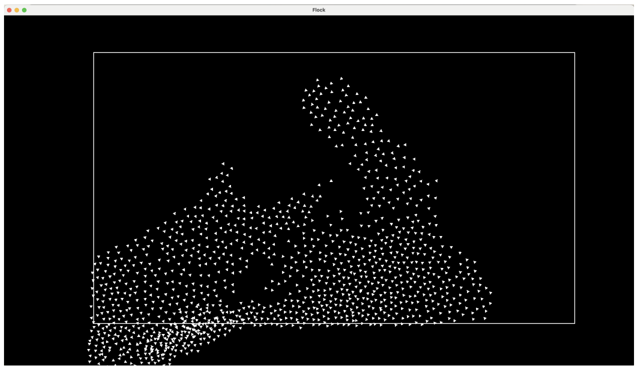
Figure 1. Visualization using SFML

1.3. OpenMP

OpenMP (Open Multiprocessing) is an API multi-platform for the creation of parallel application on shared memory system. OpenMP provides several compilation directives useful for parallel processing. The directive we used in the code is the one to parallelize the for loop that contains the update method of a boid.

```
#pragma omp parallel for
```

With this directive we can divide the loop iterations among the available threads, in such a way as to divide the workload for each thread and execute it in parallel. At the end of the loop, OpenMP automatically waits for all threads to complete execution,this mechanism is called implicit barrier.

2. Code walk-through

In this section, we will see al the C++ code implementation. The approach we used for both implementations was to use an Array of Structures (AOS), for this reason the flock element contains a vector of boids and initially allocates them with random positions within the screen. As we said previously the code is based on three main rules let's see them in more detail:

- **Cohesion** refers to the behavior of a boid that causes it to move towards the center of the mass of nearby boids. The goal of this rule is to keep the flock together and prevent individual boids from drifting too much apart. Each boid looks at its neighbors within a defined radius called **Visual Range** and calculates the average position of those neighbors, then adjusts its movement toward that average position.

$$\mathbf{F}_{\text{cohesion}} = \mathbf{b}_{\text{avg}} - \mathbf{b}_{\mathbf{i}} \qquad (1)$$

```
void Boid::cohesion(std::vector<Boid>
    flock){
    float xpos_avg = 0;
    float ypos_avg = 0;
    int neighboring_boids = 0;
    for(auto otherboid:flock){
        float dx =
            x-otherboid.get_x();
        float dy =
            y-otherboid.get_y();
        if (&otherboid != this &&
            abs(dx) < visualrange &&
            abs(dy)<  visualrange){
            float squareddistance =
                dx*dx +dy*dy;
            if (squareddistance <
                visualrange *
                visualrange) {
                xpos_avg +=
                    otherboid.get_x();
                ypos_avg +=
                    otherboid.get_y();
                neighboring_boids +=
                    1;
            }
        }
    }
    if(neighboring_boids > 0){
        xpos_avg =
            xpos_avg/neighboring_boids;
        ypos_avg =
            ypos_avg/neighboring_boids;
        Vx += (xpos_avg - x) *
            centeringfactor;
        Vy += (ypos_avg - y) *
            centeringfactor;
    }
}
```

- **Separation** is a behavior designed to avoid collisions and overcrowding among the boids. When a boid detects that there are other boids nearby, within a certain radius called **Protected Range**, it moves away from them to maintain a safe distance. This prevents boids from crashing into each other and helps maintain a natural spread within the flock.

$$\mathbf{F}_{\text{separation}} = \sum_{\text{nearby } b_j} \frac{1}{\text{distance}(\mathbf{b_i}, \mathbf{b_j})} \cdot (\mathbf{b_i} - \mathbf{b_j}) \tag{2}$$

```cpp
void Boid::separation(std::vector<Boid> flock){
    float close_dx = 0;
    float close_dy = 0;
    for (auto otherboid:flock) {
        float dx = x - otherboid.get_x();
        float dy = y - otherboid.get_y();
        if (&otherboid != this && abs(dx) < visualrange && abs(dy) < visualrange){
            float squareddistance = dx*dx + dy*dy;
            if (squareddistance < protectedrange * protectedrange){
                close_dx += x - otherboid.get_x();
                close_dy += y - otherboid.get_y();
            }
        }
    }
    Vx += close_dx * avoidfactor;
    Vy += close_dy * avoidfactor;
}
```

- **Alignment** causes each boid to steer towards the avarage direction of its nearby neighbors. A boid will try to match its speed and direction to that of the boid around it, in the **Visual Range**. This behavior helps the flock move cohesively in the same general direction without any uneven motion or collision.

$$\mathbf{F}_{\text{alignment}} = \mathbf{v}_{\text{avg}} - \mathbf{v_i} \tag{3}$$

```cpp
void Boid::align(std::vector<Boid> flock){
    float xvel_avg = 0;
    float yvel_avg = 0;
    int neighboring_boids = 0;
    for (auto otherboid:flock){
        float dx = x - otherboid.get_x();
        float dy = y - otherboid.get_y();
        if(&otherboid != this && abs(dx) < visualrange && abs(dy) < visualrange){
            float squareddistance = dx*dx + dy*dy;
            if(squareddistance < visualrange * visualrange){
                xvel_avg += otherboid.get_vx();
                yvel_avg += otherboid.get_vy();
                neighboring_boids += 1;
            }
        }
    }
    if(neighboring_boids > 0){
        xvel_avg = xvel_avg/neighboring_boids;
        yvel_avg = yvel_avg/neighboring_boids;
        Vx += (xvel_avg-Vx) * matchingfactor;
        Vy += (yvel_avg-Vy) * matchingfactor;
    }
}
```

2.1. Parameters

Careful tuning of parameters is essential in achieving realistic and desired flocking behavior. Each parameter plays a significant role in shaping the interactions and movement of the boids. Additionally different parameter sets are used for the sequential and parallel implementations to balance performance. Let's look at each parameter specifically:

- **Visual range** defines the maximum distance within which a boid can perceive its neighbors. Larger value result in more cohesive flocking, beacuse each boid can interact with more neighbors, while smaller values lead to more localized behavior.

- **Protected range** sets the minimum distance a boid tries to maintain from others to avoid collisions. A small protected range allows boids to cluster tightly, while a larger makes the flock more dispersed.
- **Matching factor** determines the strength of the alignment rule. Higher values make the boids align their velocities quickly, creating a more synchronized movement, while lower values result in slower alignment.
- **Centering factor** controls how strongly a boid moves toward the center of its neighbors. A higher value leads to tightly packed groups, while a lower values results in looser flocking
- **Avoid factor** regulates the repulsion strength within the protected range. Larger values create stronger avoidance behavior, preventing collisions more aggressively, while smaller values allow boids to come closer.
- **Turn factor** specifies how strongly a boid turns when approaching the VGAedges of the simulation area. Higher values keep the boids well within the boundaries, while lower values may allow them to stray near the VGAedges.
- **Minspeed** and **Maxspeed** define the range of allowed speeds for boids. Lower speeds result in slower and more relaxed movement, while higher speeds produce faster and more dynamic flocking behavior

For the parallel implementation, smallers values for factor like matching factor, centering factor and avoid factor are used to ensure stable and realistic behavior, while the sequential version uses higher values to maintain visual clarity. Adjusting these parameters apporpriately is critical to simulate the desired flocking dynamics and balance the computational efficency of both implementations.

```
if(parallel){
    visualrange = 55.0f;
    protectedrange = 8.0f;
    matchingfactor = 0.01f;
    centeringfactor = 0.0006f;
    avoidfactor = 0.05f;
    turnfactor = 0.05f;
    minspeed = 3.0f;
    maxspeed = 6.0f;
}
else{
    visualrange = 60.0f;
    protectedrange = 15.0f;
    matchingfactor = 0.1f;
    centeringfactor = 0.0008f;
    avoidfactor = 0.05f;
    turnfactor = 0.5f;
    minspeed = 3.0f;
    maxspeed = 6.0f;
}
```

2.2. Bias

To make the movement of the boids less predictable and more natural, a bias is applied to their direction. This bias add a small deviation to their velocity, simulating the slight unpredictability of a flocking behaviors. From Couzin *et al*. [2]: we divide the boids into two main groups, scout 1 and scout 2, based on which group the boid belongs to it will be biased to the left or to the right.

- **Dynamic bias**, the bias value *biasval* is adjusted incrementally during each simulation step. The value can either increase or decrease by a defined amount *biasincrement*, with an upper limit set by *maxbias*. This gradual change introduces evolving variations in the movement of the boids, ensuring that their paths appear fluid rather than overly deterministic.
- **Static bias**, this implementation uses a fixed value for *biasval*, providing a constant level of randomness to the movement. This approach ensures some variability while maintaining consistent behavior throughout the simulation. It is simpler to implement but lacks the evolving dynamics of the dynamic bias version.

```
//Dynamic bias
float biasval = 0.001;
float maxbias = 0.01;
float bias_increment = 0.0004;
if(scout){// biased to right of the
 ↪  screen
    if(Vx > 0){
        biasval = std::min(maxbias,
         ↪  biasval+bias_increment);
    }else{
        biasval =
         ↪  std::max(bias_increment,
         ↪  biasval-bias_increment);
    }
}else{// biased to left of the screen
    if(Vx < 0){
        biasval = std::min(maxbias,
         ↪  biasval + bias_increment);
    }else{
        biasval =
         ↪  std::max(bias_increment,
         ↪  biasval - bias_increment);
    }
}
//Velocity update
if(scout){ //Biased to the right
    Vx = (1 - biasval) * Vx + (biasval *
     ↪  1);
}else{ //Biased to the left
    Vx = (1 - biasval) * Vx +(biasval *
     ↪  (-1));
}
```

2.3. Sequential implementation

The sequential implementation simply iterates over each boid within the flock and performs the update function. The update function contains exactly everything we talked about, once it is called it performs the cohesion, separation and alignment functions, adding a bias, limiting the boids within the window and updating the respective speeds.

```
for (auto &Boid:flock){
    Boid.update(flock);
}
```

2.4. Parallel implementation

In the parallel implementation, we parallelized the for loop responsible for updating the state of each boid. Using the *#pragma omp parallel for* directive, the update operations for different boids are distributed across multiple threads, allowing them to execute concurrently. This approach significantly improves performance by utilizing modern multi-core processors. A key aspect of this implementation is that the update function does not modify or directly access the state of other boids during its execution. This design ensures that no race conditions occur, as each thread operates independently on its assigned boid. The shared data is only read by the threads, avoiding the need for complex synchronization mechanisms.

```
#pragma omp parallel for
for (size_t i = 0; i < flock.size();
 ↪  i++){
    flock[i].update(flock);
}
```

2.5. Managing implicit dependencies

A subtle but important issue in implementing flocking simulations is the management of implicit dependencies during the update process. In both sequential and parallel implementations, the naive approach typically involves updating each boid immediately based on the current state of its neighbors. This approach introduces an implicit dependency: as the iteration progress, some boids are updated with the new values of their neighbors, while others still rely on the previous values. This can lead to inconsistencies in the simulation, as the updates are not based on a coherent snapshot of the flock's state at a specific time. To address this, instead of directly updating the state of each boid within the same structure, we calculate the new state (position, velocity) for each boid and store it in a temporary structure, referred to as the *nextflock*. This ensures that all updates for a given iteration are performed based on the same consistent snapshot of the flock's state. At the end of each update loop, the two structures *currentflock* and *nextflock* are swapped. Although this operation consists of a simple swap of pointers between two vector, it will introduces some overhead which will be visible in the test section. We therefore changed the declaration of the update function from:

```
void Boid::update(std::vector<Boid>
  ↪ &flock);
```

to

```
void Boid::update(std::vector<Boid>
  ↪ &Currentflock, Boid &updateBoid);
```

After all the computations relating to the position and speed of the boid we are going to update the Boid object *updateBoid* reference. This reference will then be stored in its relative position in the new structure *nextflock*. At the end of the loop, we execute the swap method of the std::vector class to exchange the two pointers. This operation is highly efficient, as it works in constant time of $O(1)$

```
#pragma omp parallel for
for (size_t i = 0; i <
  ↪ currentFlock.size(); i++){
    currentFlock[i].update(currentFlock,
      ↪ nextFlock[i]);
}
currentFlock.swap(nextFlock); //Swap
  ↪ structures
```

3. Test

To evaluate the parallel implementation over the sequential one, two distinct tests were conducted to analyze the performance.

- **Thread Scalability Evaluation**, aimed to asses the impact of increasing the number of threads while maintaining a fixed number of boids.
- **Boids Scalability Evaluation**, the focus shifted to evaluating performances variations by altering the number of boids moving in a flocking behavior.

3.1. Speedup

To measure the performance of the parallel implementation, we calculated the speedup by comparing the execution time of the parallel and sequential version of the program. In our case we focused on the update phase of the flock, as this is the part of the code that was parallelized. In depth, we measured the time required to update all the boids in the flock during each iteration, for a total of 1000 iteration. These measurements were collected in separate runs for the sequential and parallel version of the program. To calculate the speedup, we used the formula:

$$\text{Speedup} = \frac{\text{Time(Sequential)}}{\text{Time(Parallel)}} \tag{4}$$

By recording times for both implementations over multiple iterations, we obtained a dataset that allowed us to compute the average speedup. This approach provides reliability and consistency of the data.

3.2. Thread scalability

This test is conducted with 1200 boids. As we expected the execution time drop from over 14 milliseconds to under 2 milliseconds in the parallel version with 32 threads (Figure 2). This performance improvement aligns with expectations, as parallel execution allows for concurrent processing of boids, leading to a significant time saving compared to sequential execution. Based on the measured times, the achieved speedups ar promising, reaching up to 4.53 on 32 threads (Figure 3).

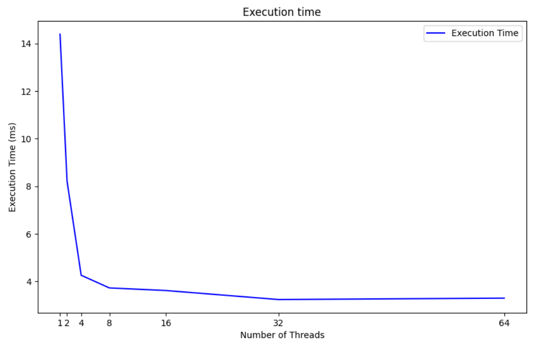

Figure 2. Execution time vs Thread number

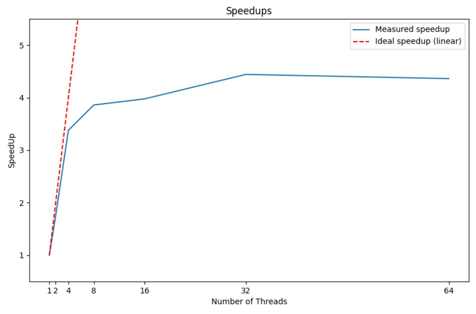

Figure 3. Speed up vs Thread number

3.3. Boids scalability

In the second test, the number of boids varied from 100 to 3200, while the number of threads ranged from 1 to 64. As expected the execution time increases from 0.13 milliseconds to 100.8 milliseconds in the sequential version as the number of boids increases (Figure 4). On the other hand, in the 8-threads parallel version, the execution time ranges from 0.14 milliseconds to 27.1084 milliseconds, demonstrating a performance increase despite the variation number of the boids. This confirms the effectiveness of parallelization in improving performance, even with varying workloads and thread counts. Then we can see related speedups (Figure 5), as the number of threads increases, the speedup values generally rise, indicating that the parallel execution leads to significant reductions in execution time compared to the sequential version. For instance, with 8 threads, the speedup values range from 3.5 to 4.5, illustrating a substantial enhancement in efficiency.

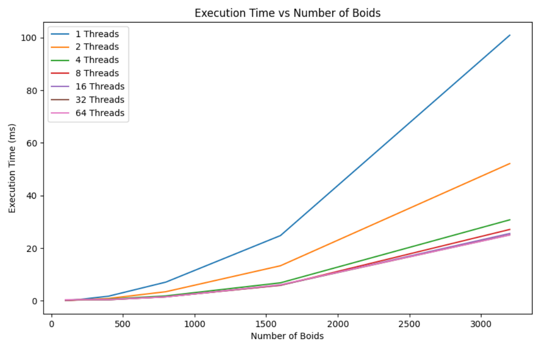

Figure 4. Execution time vs Boids number

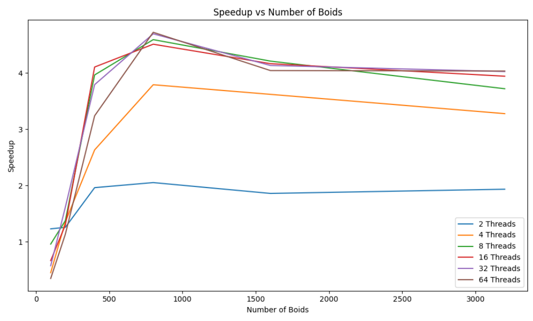

Figure 5. Speedup vs Boids number

3.4. Dependencies free

We have considered the speedup and execution time also in the version where we take into account the implicit dependency between the boids. As we said previously, the use of two different structures together with the swap between the vectors causes an overhead that we can clearly see through the two graphs below.

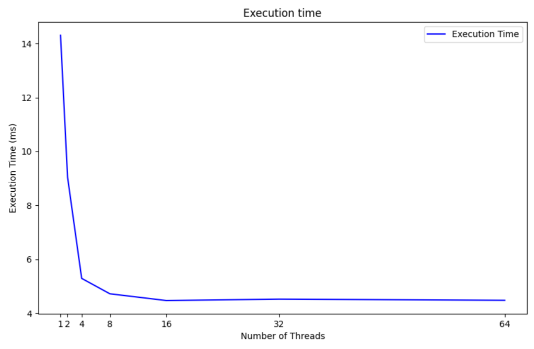

Figure 6. Execution time dependencies free

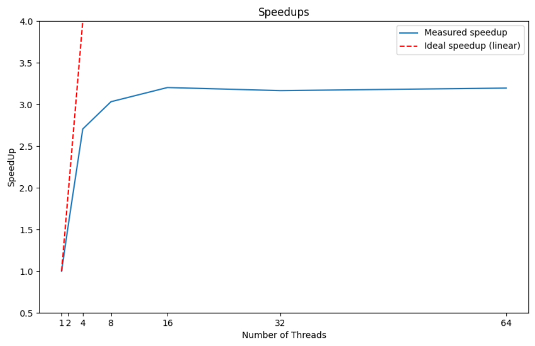

Figure 7. Speed up dependencies free

We can see how the execution times in this case do not fall below 4 milliseconds while on the other hand in the previously analyzed case the execution time, as the number of threads increased, reached values much lower than 4 milliseconds (Figure 6). We can notice the same thing for the speed up, in this case it reaches a stable value above 16 threads but never exceeds the value of 3.5, unlike the previous case where it was a value well over 4 (Figure 7).

4. Conclusion

In conclusion, the analysis of the speedup shows how parallelization can significantly reduce execution time when the number of threads increases. However, the results highlight that the speedup is not perfectly linear due to factors like thread management overhead and hardware limitations. While the performance improves with more threads, the gains become smaller after a certain point. This demonstrates the importance of balancing the number of threads and understanding the limitations of the system to achieve optimal performance.

References

[1] C. W. Reynolds, ”Flocks, Herds and Schools: A Distributed Behavioral Model,” *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH ’87)*, ACM, 1987, pp. 25–34. Available: `https://doi.org/10.1145/37401.37406`.

[2] I. D. Couzin, J. Krause, N. R. Franks, and S. A. Levin, ”Effective leadership and decision-making in animal groups on the move,” *Nature*, vol. 433, no. 7025, pp. 513–516, 2005. Available: `https://doi.org/10.1038/nature03236`.