Parallel Boids Simulation Report

Michael Bartoloni
Matricola 7152862

Abstract

The purpose of this project is an exercise in parallel programming techniques using the OpenMP framework for C++. The project features the implementation of a simulation algorithm based on birds' flocking behavior, the development and optimization of a parallel version, and subsequent testing for potential benefits in execution times.

1. Introduction

The implementation of the algorithm follows the description in the given website. The code is available in this Github repository. Given a number of boids (bird-oid objects), the simulation processes the coordinated movement of the flock, adjusting the parameters of each boid iterating across all of them.
To test for advantages in a parallel processing approach, a sequential and parallel (using the OpenMP framework) version of the algorithm was benchmarked and compared.
The first part of the benchmark tests the structures used for representing the data, finding the best approach between *Array of Structures* and *Structure of Arrays*. Once the optimal structure is found, a multithreaded version of the algorithm is benchmarked to evaluate how the execution time varies when deploying an increasing number of threads.
As discussed in the results section, there is in fact an optimal number of threads, based on the machine that runs the algorithm.
In later sections, we will go into details with the structure of both versions of the algorithm and how the benchmark is performed (for brevity, all AoS versions of the relevant code sections are shown, while for SoA only the structures are introduced).

1.1. A brief recall on bird-oid objects

Boids simulate flocking behavior through a series of parameters [Source code 1], which dictate three key behaviors:

- Separation: avoiding running into other boids
- Alignment: matching the velocity of nearby boids

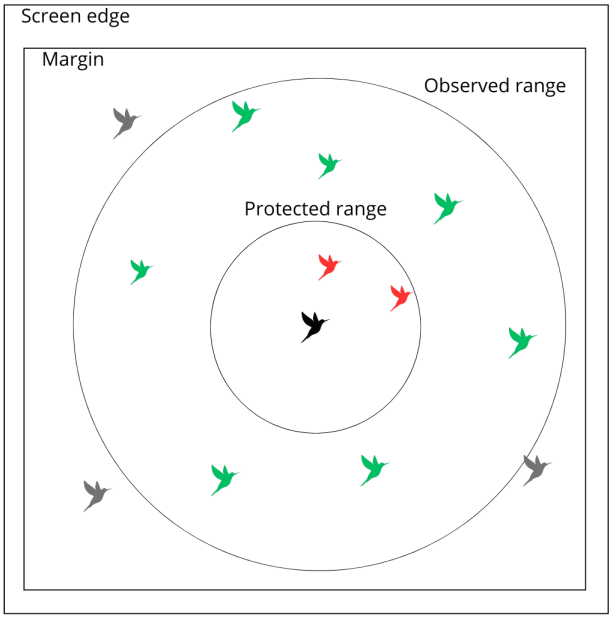

Figure 1. Boid perspective visualization.

- Cohesion: moving towards the center of mass of the observed flock

Also, we need to account for screen edges, correcting the trajectory when approaching them. In [Figure 1], a simple schema for the observations of a boid is visualized.

```
constexpr float TURN_FACTOR = 0.2f;
constexpr float MAX_SPEED = 6.0f;
constexpr float MIN_SPEED = 3.0f;
constexpr float CENTERING_FACTOR = 0.005f;
constexpr float AVOID_FACTOR = 0.015f;
constexpr float PROTECTED_RANGE = 15.0f;
constexpr float MATCHING_FACTOR = 0.01f;
constexpr float VISUAL_RANGE = 60.0f;

constexpr int WINDOW_WIDTH = 720;
constexpr int WINDOW_HEIGHT = 480;
constexpr int TOP_MARGIN = 380;
constexpr int BOTTOM_MARGIN = 100;
constexpr int LEFT_MARGIN = 100;
constexpr int RIGHT_MARGIN = 620;
```

Source code 1. Constants for trajectory manipulation

1.2. The structures

Each boid is represented by a structure that carries its two-dimensional position and speed, and some measures calculated through an iteration (like the average position or speed of neighboring boids) are stored in a dedicated structure, as shown in [Source code 2, 3, 4, 5].
Note that the Parameters struct might not be necessary to update a single boid, but since each boid update must be synchronous with the other boids (after all, what we are computing is a frame of the simulation), we first need to calculate the update, and then apply it later in a global synchronous update; hence why we need to store the update during the execution.
Both structures are padded in every version of the benchmark, to keep the baseline consistent with the parallel optimization and possibly gain something in terms of execution efficiency.

```
struct alignas(CACHE_LINE_SIZE) PadBoid {
    float x, y;    // Position
    float vx, vy;  // Velocity
    char pad[ (CACHE_LINE_SIZE > sizeof(float)
    *4) ? (CACHE_LINE_SIZE - sizeof(float)*4) :
     1 ];
};
```

Source code 2. AoS version of the boids' structure

```
struct alignas(CACHE_LINE_SIZE) PadParameters {
    float close_dx, close_dy;
    float avg_vx,    avg_vy;
    float avg_x,     avg_y;
    float neighboring_count;
    char pad[ (CACHE_LINE_SIZE > sizeof(float)
    *7) ? (CACHE_LINE_SIZE - sizeof(float)*7) :
     1 ];
};
```

Source code 3. AoS version of the boids' update structure

```
struct alignas(CACHE_LINE_SIZE) PadBoidsList {
    float x[NUM_BOIDS];
    char pad1[ (CACHE_LINE_SIZE > sizeof(float)
    *NUM_BOIDS) ? (CACHE_LINE_SIZE - sizeof(
    float)*NUM_BOIDS) : 1 ];
    float y[NUM_BOIDS];
    char pad2[ (CACHE_LINE_SIZE > sizeof(float)
    *NUM_BOIDS) ? (CACHE_LINE_SIZE - sizeof(
    float)*NUM_BOIDS) : 1 ];
    float vx[NUM_BOIDS];
    char pad3[ (CACHE_LINE_SIZE > sizeof(float)
    *NUM_BOIDS) ? (CACHE_LINE_SIZE - sizeof(
    float)*NUM_BOIDS) : 1 ];
    float vy[NUM_BOIDS];
    char pad4[ (CACHE_LINE_SIZE > sizeof(float)
    *NUM_BOIDS) ? (CACHE_LINE_SIZE - sizeof(
    float)*NUM_BOIDS) : 1 ];
};
```

Source code 4. SoA version of the boids' structure

```
struct alignas(CACHE_LINE_SIZE)
    PadParametersList {
    float close_dx[NUM_BOIDS];
    char pad1[ (CACHE_LINE_SIZE > sizeof(float)
    *NUM_BOIDS) ? (CACHE_LINE_SIZE - sizeof(
    float)*NUM_BOIDS) : 1 ];
    float close_dy[NUM_BOIDS];
    char pad2[ (CACHE_LINE_SIZE > sizeof(float)
    *NUM_BOIDS) ? (CACHE_LINE_SIZE - sizeof(
    float)*NUM_BOIDS) : 1 ];
    float avg_vx[NUM_BOIDS];
    //...
};
```

Source code 5. SoA version of the boids' update structure

1.3. The algorithm

Since OpenMP is used, the structure of the code does not vary dramatically, and we can generally introduce it here, as it will be common for both versions.
For each iteration, the computation resides in two main loops:

- A double loop, where for each boid its neighbors are found and the updates for the boid's parameters are computed (this part reads data from many different addresses) [6]
- A loop for updating each boid with its new parameters (this part is mainly writing) [7]

When these two operations are completed, the Parameters structure is reset and another iteration can start.

```
void aosGetParameters(PadBoid* boids,
    PadParameters* parameters) {
    for (int i = 0; i < NUM_BOIDS; ++i) {
        parameters[i].reset();
        for (int k = 0; k < NUM_BOIDS; ++k) {
            if (i == k) continue;

            float dx = boids[i].x - boids[k].x;
            float dy = boids[i].y - boids[k].y;

            if (std::abs(dx) < VISUAL_RANGE &&
    std::abs(dy) < VISUAL_RANGE) {
                float squared_dist = dx * dx +
    dy * dy;

                if (squared_dist <
    PROTECTED_RANGE * PROTECTED_RANGE) {
                    parameters[i].close_dx +=
    dx;
                    parameters[i].close_dy +=
    dy;
                } else if (squared_dist <
    VISUAL_RANGE * VISUAL_RANGE) {
                    parameters[i].avg_x +=
    boids[k].x;
                    parameters[i].avg_y +=
    boids[k].y;
                    parameters[i].avg_vx +=
    boids[k].vx;
                    parameters[i].avg_vy +=
    boids[k].vy;
```

```
                    parameters[i].
    neighboring_count += 1;
                }
            }
        }
    }
}
```

Source code 6. Parameters gatherer method

```
void aosUpdate(PadBoid* boids, PadParameters*
    params) {
    for(int i = 0; i < NUM_BOIDS; ++i) {
        if (params->neighboring_count > 0) {
            //average some parameters
            //centering and avoiding boids that
     are too close
        }

        if (boids->y > TOP_MARGIN) boids->vy -=
     TURN_FACTOR;
        if (boids->x > RIGHT_MARGIN) boids->vx
    -= TURN_FACTOR;
        if (boids->x < LEFT_MARGIN) boids->vx
    += TURN_FACTOR;
        if (boids->y < BOTTOM_MARGIN) boids->vy
     += TURN_FACTOR;

        float speed = std::sqrt(boids->vx *
    boids->vx + boids->vy * boids->vy);
        if (speed > MAX_SPEED) {
            boids->vx = (boids->vx / speed) *
    MAX_SPEED;
            boids->vy = (boids->vy / speed) *
    MAX_SPEED;
        } else if (speed < MIN_SPEED) {
            boids->vx = (boids->vx / speed) *
    MIN_SPEED;
            boids->vy = (boids->vy / speed) *
    MIN_SPEED;
        }

        boids->x += boids->vx;
        boids->y += boids->vy;
    }
}
```

Source code 7. Update method

2. Sequential Benchmark

The sequential version of the algorithm features a comparison between Array of Structures and Structure of Arrays, estabilishing a baseline for the parallel benchmark.
The benchmark consists of calculating the execution time of a single iteration of the algorithm for 40000 boids, averaging the results on 100 runs.
This test explores the impact of the memory access patterns of the algorithm: a mix of multicoordinate operations, such as distance calculation (suited for AoS) and single coordinate operations, such as parameter sums (suited for SoA). Theoretically, we would expect the SoA version to run faster, as between two boids only one distance is calculated, but many parameters are summed.

Array of Structures	Structure of Arrays
5.4869	4.2752

Table 1. Sequential execution times (s)

2.1. Results

The results [Table 1] show a significant increase in performance for the SoA approach. This is the baseline for the parallel version and confirms the assumption that SoA operation prevalence indeed plays a role in the actual execution time, at least for a sequential setting.

3. Parallel Benchmark

The benchmark features an optimized version of the algorithm, and is run on an increasing number of threads, going up from 2 to 20, for both versions of the structures. The number of boids and averaging runs remains the same as the sequential benchmark.
We do not need to test for more threads, as the testbed features a 6 cores, 12 threads processor, so we expect the speedup to stop increasing at 12 threads and some performance drop due to overhead going over that.
Some considerations are necessary to bring out most of the potential of the parallelized versions, and have to be reasoned about. They are described in the next section.

3.1. Optimizations

OpenMP features a fairly simple framework, parallelizing the code is almost straightforward. However, there are a couple of optimizations that aim to bring out the potential of multithreading. In this section, we discuss the ones implemented for the task.

```
 #pragma omp parallel num_threads(numThreads)
    shared(aosBoids) firstprivate(aosParameters
    ) default(none)
{
    getParametersAOS(aosBoids, aosParameters);
    updateAOS(aosBoids, aosParameters);
}
```

Source code 8. Parallelized benchmark

```
void getParametersAOS(PadBoid* boids,
    PadParameters* parameters) {
    #pragma omp for schedule(static) nowait
    for (int j = 0; j < NUM_BOIDS; ++j) {
        //Same as the sequential version
    }
}

void updateAOS(PadBoid* boids, PadParameters*
    parameters) {
    #pragma omp for schedule(static)
    for(int j = 0; j < NUM_BOIDS; ++j) {
       //Same as the sequential version
    }
```

```
}
```

Source code 9. Parallelized versions of the methods

```
#pragma omp for schedule(static)
for(int i = 0; i < NUM_BOIDS; ++i) {
    aosBoids[i] = PadBoid(static_cast<float>(
    rand() % WINDOW_WIDTH), static_cast<float>(
    rand() % WINDOW_HEIGHT));
}
```

Source code 10. First touch initialization

3.1.1 Private copies and reduction

Privatizing frequently accessed resources reduces the amount of synchronization needed, at the cost of some memory.
Looking at the structure of an iteration, the Parameters structure is accessed with unpredictability, given that those accesses depend on the boids' coordinates at runtime.
This would require some critical sections in the updates to avoid race conditions, but privatizing the structure ensures no race conditions, so each thread can move on to calculate the update, cutting thread communication and, therefore, improving performance. Since we do not care about initial values of the structure, as we reset it at the start of each iteration, we use the *firstprivate* argument [Source code 8] passing the Parameters structure just to have an initialized copy.
As for the Boids structure, we can avoid privatizing it. The reason is the usage of the structure in the execution of the algorithm, as it is mainly read and only written at the end of the iteration to update the state of the boids and the writing pattern is very predictable, as each thread takes care of its unique slice of data. So we declare it as a shared variable [Source code 8].
The final consideration for what concerns using the shared Boids structure is the eventual need to use a critical section when updating the state of the shared Boids vector.
We actually don't need to. The motivation lies in the threads' organized access to the resource. If we set the OMP schedule to static [Source code 8], each thread will always access distinct and contiguous ranges of indexes, we have no race conditions in both versions. Padding the structures avoids false sharing completely in an Aos structure but leaves the SoA version vulnerable. However, even if there is some false sharing, it is negligible given that the only parts that might experience it are the neighboring indexes between two threads' slices, and in comparison to the size of the arrays at play, the percentage is too small to have a serious impact on performance (at most 0.05% of all the Boids addresses are at risk of false sharing with 40000 boids and 20 threads).

Figure 2. Plot of the execution times with growing number of threads

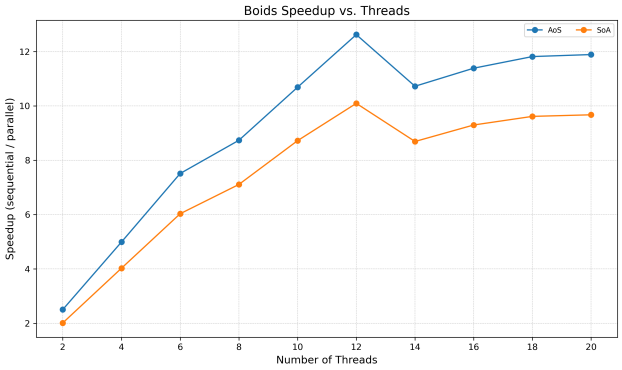

Figure 3. Plot of the execution times with growing number of threads

3.1.2 First touch initialization

Given that each thread updates its own range of indexes, it makes sense to bring each resource closer to the thread that uses it. This could potentially benefit the updating process, so we parallelize the initialization of the Boids variables [Source code 10].

3.1.3 Barriers

Since we have private variables and a final reduction, it makes sense to remove the barrier in the getParameters() method with the nowait argument [Source code 9], as the for cycle works exclusively on private variables and their effect has no impact on the shared ones. The barrier must remain in the update method for general correctness (even if in this benchmark the threads stop after a single iteration).

3.2. Results

In [Figures 2, 3], a view of execution times is shown as we increase the number of threads. As the tabular data show, the speedup is almost always superlinear for the AoS version, and almost linear for the SoA version. Not only

Number of threads	AoS Time (s)	SoA Time (s)
2	2.242	2.167
4	1.105	1.082
6	0.739	0.715
8	0.627	0.601
10	0.515	0.491
12	0.486	0.456
14	0.520	0.500
16	0.491	0.474
18	0.480	0.459
20	0.472	0.461

Table 2. Execution Times for AoS and SoA Layouts

Number of threads	AoS Speedup	SoA Speedup
2	2.447	1.973
4	4.966	3.951
6	7.425	5.979
8	8.751	7.114
10	10.654	8.707
12	11.290	9.375
14	10.552	8.550
16	11.175	9.019
18	11.431	9.314
20	11.625	9.274

Table 3. Speedups for AoS and SoA Layouts

that, but the results [Tables 2, 3] show that the optimal number of processors is 12 (as we discussed when introducing the testbed's processor), and after that, a little overhead impacts on the performance. SoA still is a little bit faster in terms of execution times, but the difference that was significant in the sequential version is greatly mitigated by deploying more threads.

4. Conclusions

It's clear that something is going on for the AoS version of the benchmark. If we look at it, it's very possible that caching is a factor: contrary to the SoA approach, a line of cache can contain a full boid, meaning there are some operations in which there are no misses when fetching data, and we can also be quite sure that 40000 boids fit into the L2 cache, leading to very fast memory access. Slicing the structures between threads enhances the phenomenon, meaning more gain in execution time. On the contrary, no boid is going to fit into a line of cache when following the SoA approach, so the benefits of parallelizing stop at a (still really good) linear rate until logical cores utilization, and from then on, the rate drops to sublinear.

The performance gain when more physical cores are available is indeed higher than the gain when deploying more threads that use shared physical cores but still consistently improves thanks to the reduced idle time in the physical cores introduced by SMT. Spawning more than 12 threads leads to some overhead that actually slows down the computation, but we can see that the overall execution time when using all the processor's threads is still less than the execution time when spawning fewer threads than it can handle.

The results of the parallel benchmark justify parallelizing it for a significant speedup in the frame computation. The speedup in this setting is very high, and the advantage of an optimized algorithm using the optimal number of threads cuts the execution time into a fraction of the sequential time, so simulation frames that require seconds can be processed in less than a second.