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.

1.1. A brief recall on bird-oid objects

Boids simulate the flocking behavior through a series of parameters, which dictate three key behaviors:

- Separation: avoiding running into other boids
- Alignment: matching the velocity of nearby boids
- Cohesion: moving towards the center of mass of the observed flock

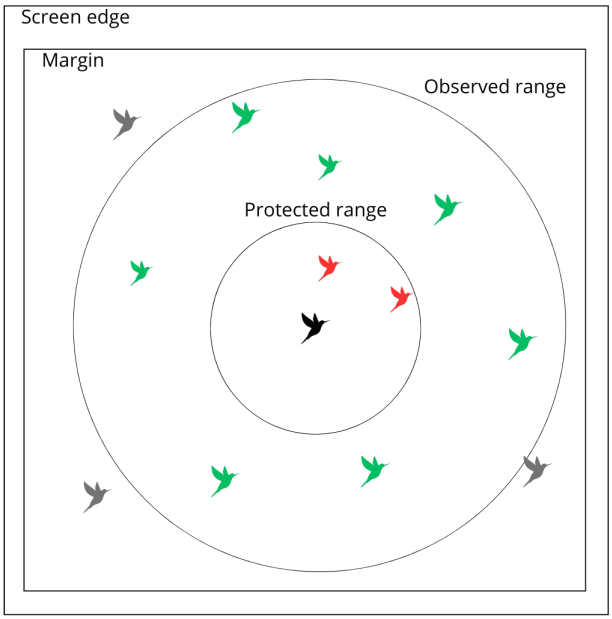

Figure 1. Boid perspective visualization.

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.

1.2. The structures

Each boid is represented by a structure that carries its two-dimensional position and speed, and the state updates calculated through an iteration are stored in the corresponding Parameters structure.
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; 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.

Array of Structures	Structure of Arrays
5.4869	4.2752

Table 1. Sequential execution times (s)

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)
- A loop for updating each boid with its new parameters (this part is mainly writing)

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

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.
This part is fundamental, as we can select an optimal structure for the parallel dataset beforehand, so that we can only focus on optimizing the algorithm.
The need for such a benchmark is to be found in 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.

2.1. Results

The results [Table 1] show a significant increase in performance for the SoA approach, which confirms the previous assumptions. This is the baseline for the parallel version.

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.
We do not need to test for more threads, as the testbed features an AMD Ryzen 3600X processor (6 cores, 12 threads), so we expect the speedup to stop increasing at 12 threads and some performance drop due to overhead going over that.

3.1. Optimizations

3.1.1 Private copies and reduction

Privatizing resources frequently accessed reduces the amount of synchronization needed, at the cost of some memory.
Looking at the structure of an iteration, the parameters are accessed with some unpredictability, given that those updates depend on the boids' coordinates.
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.
As for the Boids structure, we can also benefit from privatization. Alongside the main shared resource, a local boids copy is kept, reflecting the current frame, and each thread performs a reduction when done calculating its part of the updates.
At the end of the iteration, a parallel for loop is added for the threads to reduce their local updated boids into the shared boids. This part is the only one requiring a critical section, as we have to avoid race conditions.

3.1.2 Barriers

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

3.2. Results

In [Figure 2], a view of execution times is shown as we increase the number of threads. The speedup is sublinear. Not only is it sublinear, but the results [Table 2] show that the optimal number of processors is 12 (as we discussed when introducing the testbed's processor).

4. Conclusions

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

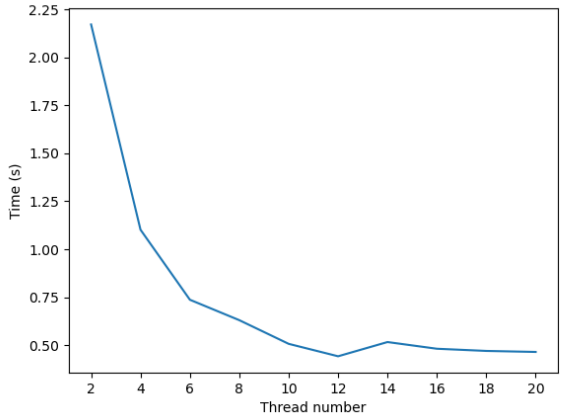

Figure 2. Plot of the execution times with growing number of threads

Number of threads	Time (s)
2	2.1709
4	1.1018
6	0.7371
8	0.6307
10	0.5074
12	0.4427
14	0.5167
16	0.4819
18	0.4705
20	0.4652

Table 2. Execution times for each thread

spawning fewer threads than it can handle.

The results of the parallel benchmark justify parallelizing it for a significant speedup in the frame computation. Even if the speed-up is sublinear, the advantage of an optimized algorithm using the optimal number of threads cuts the execution time into a fraction of the sequential time.

Looking at the execution time for 6 threads, we can see how little the improvements are when using more threads. Although it is significant, one could expect more improvement when increasing the number of threads (when the machines running them could fit more). This could be explained with SMT, as 6 is the number of physical cores and each one has two logical cores.

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 performance thanks to the reduced idle time in the physical cores introduced by SMT. Note that this comparison makes sense when the number of threads increases by the same factor, as shown in [Table 3].

Increase factor	Starting threads number	Speedup
1.5	4	~1.49
1.5	8	~1.43
2	1	~1.97
2	2	~1.97
2	4	~1.75
2	6	~1.67
3	4	~2.49
3	6	~1.57

Table 3. Speedup when multiplying the number of threads by some factor