First Assignment: Boids parallelization and performance evaluation with OpenMP

Author: Giacomo Orsucci
E-mail address:
giacomo.orsucci@edu.unifi.it

Lecturer: Marco Bertini
Course:
Parallel Computing

Abstract

Nowadays, is more than ever necessary to parallelize code execution to fully exploit modern multi-core processors and to reduce significantly the execution time of computationally intensive tasks. For example, simulating realistic flocking behavior in computer graphics and artificial life applications is a computationally intensive task. The purpose of this assignment is to parallelize the Boids algorithm using OpenMP with C/C++ and evaluate its performance. The Boids algorithm simulates the flocking behavior of birds through simple rules applied to individual agents. By leveraging parallel computing techniques, we aim to enhance the sequential simulation execution time. In order to achieve this, we'll be explored different versions of the algorithm, including the AOS (Array of Structures) and SOA (Structure of Arrays) versions, and the benefits that can be obtained by introducing padding and SIMD instructions. This report details the implementation process, performance metrics, and analysis of the results obtained from the parallelized version of the Boids algorithm with a complete but as compact as possible approach. After a brief introduction can be found a detailed and experimental oriented description of all the parallel implementations tested.

Future Distribution Permission

1. Introduction

1.1. Boids algorithm

The Boids algorithm, developed by Craig Reynolds in 1986, is a computational model that simulates the flocking behavior of birds and other social animals. The algorithm is based on three simple rules that govern the movement of individual agents, known as *boids*, within a simulated environment. These rules are:

- **Separation**: Boids steer to avoid to be too close to local flockmates.
- **Alignment**: Each boid attempts to match the velocity of other boids inside its visible range.
- **Cohesion**: Each boid steers gently toward the center of mass of other boids within its visible range.

All this aspects are combined to produce realistic flocking behavior, where boids move in a coordinated manner while avoiding collisions with each other, but are largerly parametrized to allow different types of flocking behavior and adapt to different environments.

More details about the Boids algorithm (including the pseudocode) can be found at [2]. The sequential version implemented derives from the translation and adjustment of the pseudocode provided in [2].

1.2. Why parallelizing Boids?

I decided to parallelize the Boids algorithm because it is a computationally intensive task, especially when simulating large flocks of boids. Moreover, the algorithm is inherently parallelizable, as each boid's behavior can be computed independently of others and the flock behavior can be checked implementing the graphical, that has not been parallelized but used to check the correct code implementation in addition to sanitiy checks.

1.3. Test Machine specifications

The tests have been performed on my personal laptop with the following specifications:

- **CPU**: AMD Ryzen 7 3700U, 4 cores, 8 threads, base clock 2.3 GHz, max boost clock 4.0 GHz.
- **Cache**:
  - L1 Cache: 96 KB per core
  - L2 Cache: 512 KB per core
  - L3 Cache: 4 MB shared.
- **RAM**: 20 GB DDR4 (1 x 4 GB SODIMM DDR4 + 1 x 16 GB Row of Chips DDR4) at 2400 MHz.
- **Operating System**: Ubuntu 22.04 LTS.
- **Compiler**: GCC 11.4.0.
- **OpenMP version**: 5.2.

2. Implementations and experiments details

In this section are reported all the details about the different implementations, the meausurement methodology adopted and the results obtained.

2.1. Measurements details

- The measurements have been performed running the simulation for a certain number of frames, where a frame is defined as the update of all the boids positions and velocities according to the Boids rules. In other words, according to the code provided in [3], a frame is a complete iteration of the external loop in the main function:

```
for each boid:
    ...
    for each other boid:
        ...
```

- Everything concerning the graphical part is not contemplated in the measurements, as the focus is on the performance of the Boids algorithm itself. The graphical part is used only to verify the correctness of the implementation and to perform indirect sanity checks.
- The duration time measures only the time taken to update a frame as defined above and is measured using `std::chrono::high_resolution_clock::now()`.
- Each experiment has been repeated 6 times, the first execution discarded and the average value obtained on the other 5 has been considered and used as the final measure.
- The number of frames simulated for each experiment has been fixed to 300.
- The increasing number of boids simulated are: 1500, 3000, 6000, 9000, 12000 because proven to be significant for our purposes thanks to some pretests not here reported because not relevant.
- Has been chosen a static scheduling because the workload is uniformly distributed among the threads, so a dynamic scheduling would have introduced unnecessary overhead.
- The number of threads used are: 1, 2, 4, 8, to cover all the possible configurations of the test machine. Tests with a number of threads greater the CPU threads would have introduced unnecessary context switching and overhead that would have distorted the measurements.

2.2. Implementations details

2.2.1 Graphics

The Graphic representation of the boids, fundamental to verify the correctness of the implementation, is implemented using the SFML library [1] that is not parallelizable and so is not taken into account in experiments. Moreover, the graphical part is not relevant for the purpose of this assignment, having the only goal to verify the correctness of the implementation.

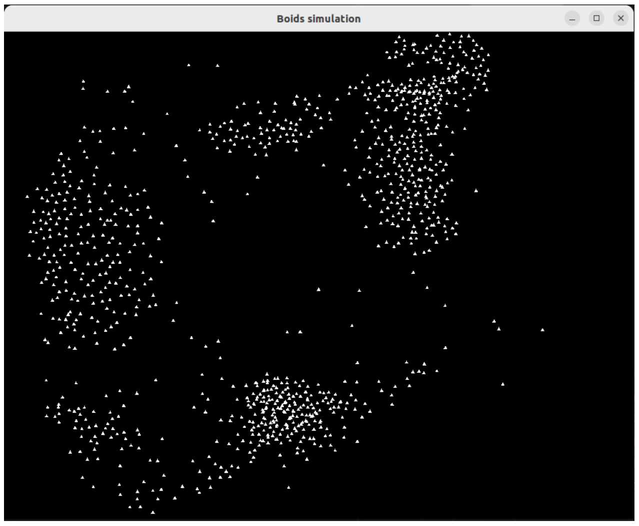

Figure 1. Example of graphical representation of the Boids simulation with 1000 boids.

2.2.2 Sanity checks and profiling

As mentioned before, the graphical part is used to verify visually the correctness of the implementation, but in addition some sanity checks have been implemented to ensure that the parallel implementations are not affected by race conditions or other issues that could compromise the correctness of the simulation. These sanity checks are included in debug mode (as visible in CMakeLists.txt in [3]):

```
-fsanitize=address,leak,undefined
```

where these flags, respectively, are:

- **address**: to detect memory errors such as buffer overflows, use-after-free, and memory leaks.
- **leak**: to detect memory leaks specifically, helping to identify memory that has been allocated but not properly deallocated.
- **undefined**: to detect undefined behavior in the code, such as null pointer dereferencing and integer overflows.

2.2.3 General optimizations

All the versions implemented share some general optimizations that have been applied to code to improve performance (in benchmark mode):

- **-O3**: compilation flag that tells the compiler to optimize as much as possible the code unrolling loops and inlining functions, where possible.
- **-march=native**: enables all the instruction sets supported by the host machine, allowing the compiler to generate optimized code for the specific architecture. Probably, the cost of this flag activation is the loss of compatibility with older processors that do not support the same instruction sets.
- **-mfma**: enables the use of Fused Multiply-Add instructions, which allows to perform multiplication and addition in a single instruction.
- **-ffast-math**: enables aggressive floating-point optimizations that may violate strict IEEE or ISO rules but can improve performance.
- **-fno-math-errno**: disables the setting of the global `errno` variable for floating-point math functions, which can improve performance translating, for example, *sqrt* in a single assembly instruction.
- **-mavx2**: enables the use of AVX2 instruction set, which provides advanced vector extensions for SIMD operations.

2.2.4 AOS parallel implementation

The Array of Structures (AOS) sequential implementation is the baseline for all the other implementations. The AOS sequential version is simply the parallel version with num threads = 1 because has been empirically proven the equivalence with the "real" sequential implementation. The AOS implementation, as suggested by the name, manages an array of Boids structure. So, in this case we'll have a structure defining a Boid with all its properties (position, velocity, etc.):

```
    struct Boid {
      float x, y;
      float vx, vy;
};
```

and an array of these structs:

```
std::vector<Boid> boids(N);
```

2.2.5 SOA parallel implementation

A different approach to data organization is the Structure of Arrays (SOA) implementation. In this case, instead of having an array of structures, we have separate arrays for each property of the boids. For example, we would have one array for the x positions, another for the y positions, another for the x velocities, and so on:

```
    struct Boids {
      float *x, *y;
      float *vx, *vy;
};
```

And we no longer have an array of Boid structs.

2.2.6 OpenMP directives

As visible in [3] we have a parallel implementation both for the AOS version and for the SOA version. In addition, both are reimplemented using OpenMP SIMD directives (more below). But what all the parallel versions have in common are the following directives:

- `#pragma omp parallel default(none) shared()`

  to create a parallel region where all the threads are spawned. The `default(none)` clause forces to explicitly specify the data sharing attributes of all variables used in the parallel region. This ensures a better control over the data sharing and helps to avoid unintended data sharing. The number of threads spawned is set previously with:

  ```
  omp_set_num_threads();
  ```

- `#pragma omp for schedule(static)`

  to distribute the iterations of the for loop among the threads in a static manner. This means that the iterations are divided into equal-sized chunks and assigned to each thread at the beginning of the loop execution. In this case is very appropriate because we have a uniform workload distribution among the iterations, using a dynamic schedule could introduce unnecessary overhead and interfere with cache locality. To be fair, in this very specific case the different has been empirically proven (with some pretests) not to be significant.

2.2.7 SIMD parallel implementation

SIMD instructions enables operations with expanded registers (256bit vs 32bit), this allows to perform the same operation on multiple data points simultaneously, for example 8 floats at a time with AVX2 instructions. Thus, it's immediately clear that SIMD instructions, if used properly, can significantly improve the speedup. With OpenMP, we can use:

```
#pragma omp simd
```

to instruct the compiler to vectorize the following loop using SIMD instructions. But must be noted that all or much of the speedup achievable with SIMD can be lost if the loop contains branches (if-else statements). In this case is profitable to implement branchless logic using arithmetic operations and bitwise operations to replace branches. For example, in the following snippet is reported a piece of code with a branch logic:

```
if (sqd < PROTECTED_RANGE*
    PROTECTED_RANGE) {

      close_dx += (xi - boids.x[j]);
      close_dy += (yi - boids.y[j]);
}
```

That has been rewritten in:

```
float dx = xi - boids.x[j];
float dy = yi - boids.y[j];

close_dx += dx * is_protected;
```

```
    close_dy += dy * is_protected;
```

where the first snippet contains a branch that can hinder vectorization, while the second snippet uses branchless logic to achieve the same result but with better performance.

Moreover, has been experimented also the reduction with SIMD, but in the results section we'll observe how this approach has not brought any improvement. For this reason the final code version does not include this optimization.

2.2.8 Padding use

Also the impact of padding has been evaluated in the AOS and SOA SIMD versions. Has been choosen the SIMD versions to investigate the effects on the best performing versions. Padding consists in adding extra bytes to the data structures to ensure that they are aligned in memory according to the requirements of the SIMD instructions. This can help to improve performance by reducing the number of memory accesses and improving cache locality. In this case, has been added padding to reach a size of 32 bytes for each structure in the AOS version and for each array in the SOA version. With the AOS version, has been tested an "implicit padding" forcing the data alignment:

```
    struct alignas(CACHE_SIZE) Boid {
     float x, y;
     float vx, vy;

};
```

but also in addition to a more explicit padding:

```
    struct alignas(CACHE_SIZE) Boid {
     float x, y;
     float vx, vy;
     char padding[32 - sizeof(float) * 4];
};
```

(All the results reported below).

Instead, for the SOA version, has been used only the aligned allocation in:

```
inline Boids allocate_aligned_boids(int N);
```

3. Experiments and results

In this section are reported all the plots obtained from the experiments performed and all the results are resumed in a compact way. Plots display the average speedup obtained with a different number of threads for an increasing number of boids simulated. All the experiments are executed for 300 frames. The blue line is the sequential execution with 1 thread, the base line used to calculate the speedups.

3.1. AOS vs Sequential (AOS with 1 thread)

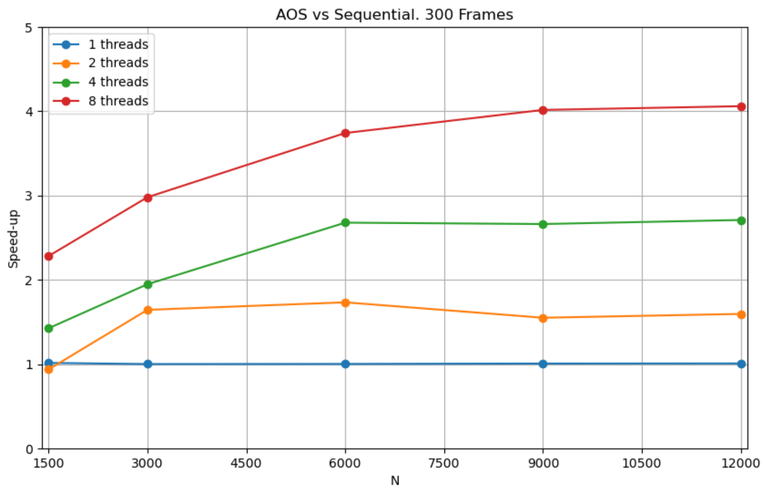

Figure 2. Speedups comparison of AOS parallel version with the sequential one.

The plot is quite self-explanatory, the AOS parallel version achieves significant speedups compared to the sequential version as the number of boids and threads increases, but with a saturation effect that limits the speedup for every number of threads. This is because we achieve better performance with more threads, but at the same time, given the AOS organization, every time a boid position is checked respect other boids, also the velocity is loaded occupying cache space unnecessarily. So, as the number of boids increases, the cache misses increase too, limiting the performance improvement achievable with more threads. Also, the branching in code contributes to limit the gains obtained from compile time optimizations.

What about using SIMD instructions?

3.2. AOS_SIMD vs Sequential

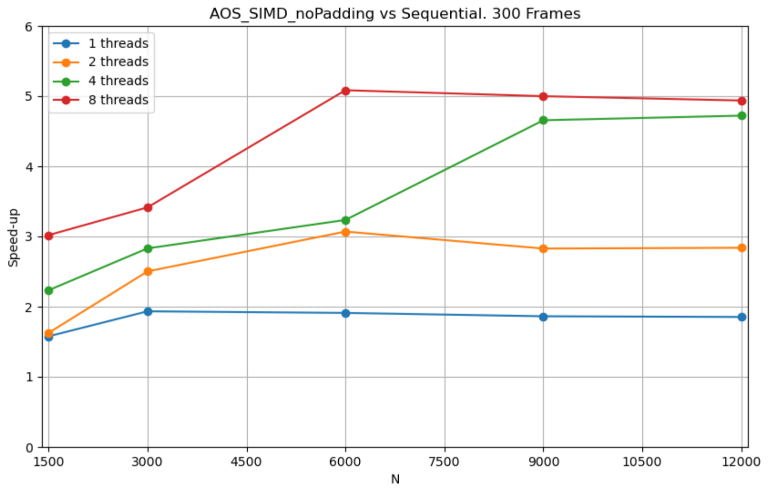

Figure 3. Speedups comparison of AOS_SIMD parallel version with the sequential one.

The SIMD contribution in speedup is evident, in fact also the baseline (1 thread) achieves a speedup compared to the sequential version. Moreover, the speedup increases more consistently with the number of threads, thanks to the SIMD instructions that allow to process multiple boids simultaneously and the implemented branchless logic (where possible) but the plateau effect is still present for the same reasons explained above. Furthermore, is noticeable that for 2 and 4 threads the differences are very little until the number of boids is below or equal 6000. Probably because the "coordination costs" for an increased number of threads in this case is not well amortise by the load until it reaches a significant mass (more than 6000 boids). The speed up bound is still given by the memory. Finally, the *_noPadding* nomenclature here and after denotes the absence of internal padding, but the presence of alignment (still a kind of padding, but is more clear defining this differentiation). What about introducing more internal padding?

3.3. AOS_SIMD_Padding vs Sequential

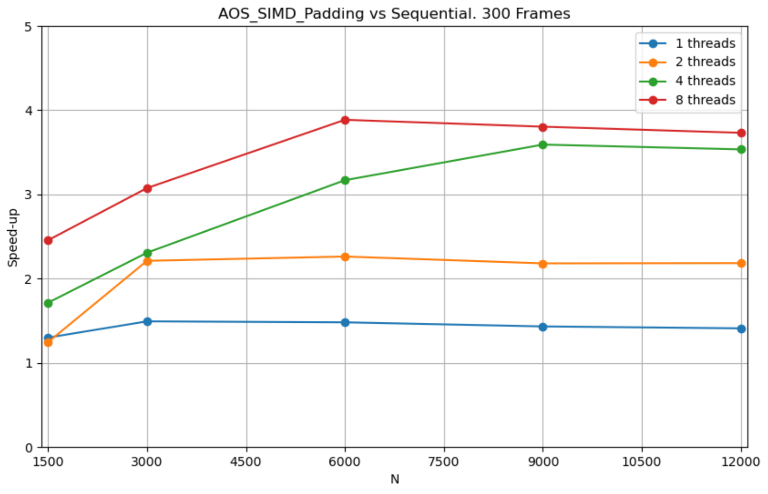

Figure 4. Speedups comparison between AOS_SIMD parallel version with padding and the sequential one.

Using Padding also inside Boid struct, worsen the performance as expected. In fact, the speedups achieved are lower than the AOS_SIMD version without padding. This is because, due to AOS organization, the padding implemented inside the Boid struct increases the size of each Boid, thus reducing the number of boids that can fit in the cache, leading to a more inefficient use of the cache. In conclusion, forcing the cache alignment is benefical but adding padding to reach a specific size is counterproductive. The last variant exeperimented is the AOS version with SIMD and reduction:

3.4. AOS_SIMD_Reduction vs Sequential

The need to try to use reduction is bornt observing that the *for* to which is applied

```
#pragma omp simd
```

contains a lot of sum parallelizable with the reduction clause. So, to assess reduction impact, to the *for* has been applied

```
#pragma omp simd reduction(+:close_dx,
    close_dy, x_avg, y_avg, xv_avg, yv_avg,
    n_neighbours)
```

with the following results:

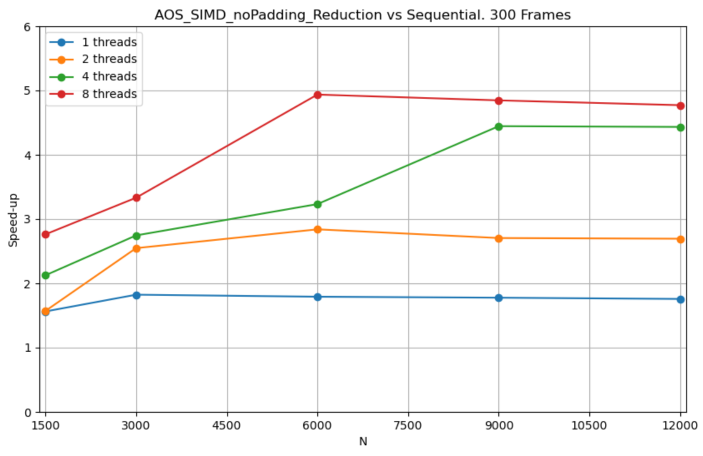

Figure 5. Speedups comparison between AOS_SIMD parallel version with Reduction and the sequential one.

As we can see, comparing the observed performance with Figure 3, introducing the reduction produced little (in worse) to no effects.

So, in relation to what has been experimented and observed, in the following plots regarding the SOA version it's possible to see how this different data organization can improve performance (improving cache locality), but without the padding (only alignment) and reduction. It is important to note that the reduction is not strictly necessary because the sanity checks have confirmed the correctness of the implementation without it and it's also possible to verify it visually executing the graphical part of the code.

3.5. SOA vs Sequential

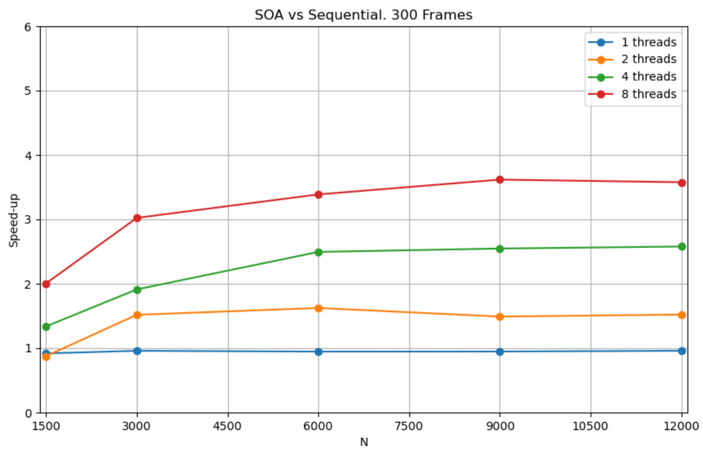

Figure 6. Speedups comparison between SOA parallel version and the sequential one.

As anticipated before, was expected that the SOA version would have achieved better performance than the AOS version, but in this plot we see a slightly worse performance with 8 threads, more precisely a 3.6 speedup vs 4.0 of the AOS version. This is because the SOA version, without SIMD instructions, does not benefit as much as from the improved cache locality as the AOS version does from its parallelization. In fact, analyzing the SIMD and branchless (in the sense that the branch have been removed where possible) version:

3.6. SOA_SIMD vs Sequential

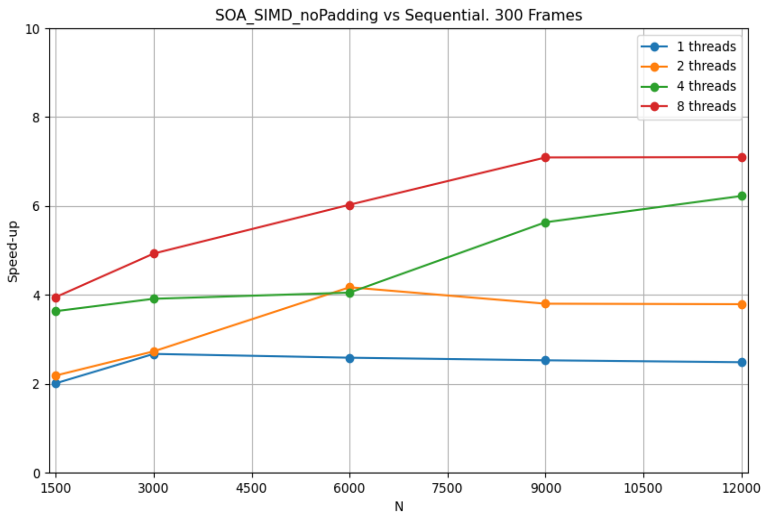

Figure 7. Speedups comparison between SOA_SIMD parallel version and the sequential one.

It's very clear how the SOA organization benefits significantly from the SIMD instructions, achieving better performance than the AOS_SIMD version in figure 3 and how this is the best parallel version observed. The performance improvement is due to the fact that the SOA organization allows to load multiple boid properties into SIMD registers more efficiently, reducing memory accesses and improving cache locality. This leads to a more efficient use of the CPU's computational resources, resulting in higher speedups. The improvement applies to all the different number of threads experimented. As already said, the *noPadding* wording indicates the presence of alignment but without internal padding. For completeness, I tried to use reduction with SIMD also with the SOA version:

3.7. SOA_SIMD_Reduction vs Sequential

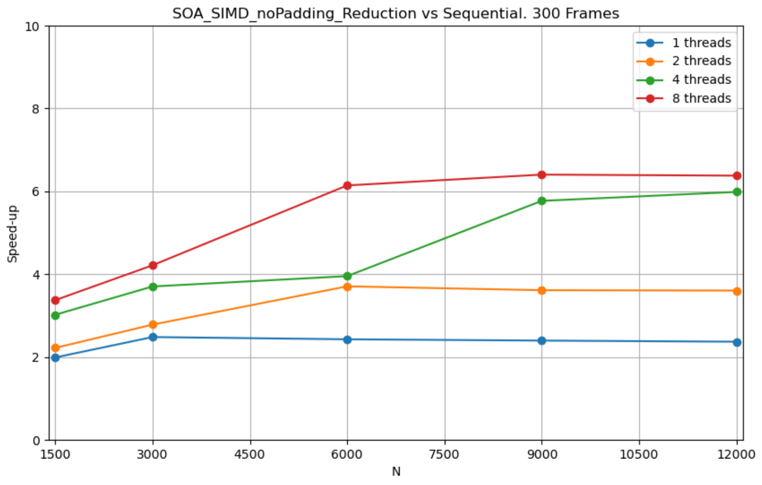

Figure 8. Speedups comparison between SOA_SIMD parallel version with Reduction and the sequential one.

Also in this case, as for the AOS version (Figure 5), the introduction of reduction has not brought any significant improvement in performance.

4. Conclusion

In conclusion, in every case tested the speedups achieved follow a sublinear trend but with significant improvements compared to the sequential version. So, generally speaking, parallelizing the Boids algorithm with OpenMP has proven to be an effective way to improve performance, especially with a number of threads equal to 8 (the maximum available on the test machine). In this report has not been tested and reported explicitly the worsen of performance with a number of threads greater than the available ones on the test machine, but has been empirically observed with pretests (and eventually, a more extendend analysis on this can be a good expansion of this work) that this happens due to context switching and overhead introduced. Thus, to obtain the best performance is important to choose the maximum number of threads available and a good load (number of boids). The best performance has been achieved with the SOA_SIMD version without internal padding (only alignment) and reduction, reaching a maximum speedup of 7 with 8 threads and 12000 boids. Therefore, using SIMD instructions reducing to the minimum the branching in code, using wisely the internal padding (or alignment) and choosing a good data organization (SOA in this case) are fundamental to achieve the best performance possible when parallelizing the Boids algorithm with OpenMP. Finally, can also be investigated more exstensively the impact of reduction and different types of scheduling (static vs dynamic) in future works, but in this specific case they have not brought significant improvements.

References

[1] Sfml library. `https://www.sfml-dev.org/`.

[2] V. H. Adams. Boids pseudocode. `https://vanhunteradams.com/Pico/Animal_Movement/Boids-algorithm.html`.

[3] G. Orsucci. Boids parallelization. `https://github.com/Giacomo-Orsucci/Boids_simulation`, 2026.