BOIDS: Analysis of speedup achieved through parallel programming on CPU

Gianni Moretti
E-mail address
gianni.moretti@edu.unifi.it

Abstract

This project explores the implementation and performance analysis of a Boids simulation in C++ with sequential and OpenMP-parallelized versions. The study compares two data layouts, Array of Structures (AoS) and Structure of Arrays (SoA), to assess their impact on performance.

Results demonstrate significant speedup with parallelization and highlight the importance of data organization for memory access efficiency. The findings underscore the value of combining parallelization and optimized data layouts in high-performance simulations. All the code used on this project can be found on https://github.com/GianniMoretti/Boids.

1. Introduction

The Boids algorithm, introduced by Craig Reynolds, simulates collective behaviors such as flocking or schooling through simple interaction rules. However, its computational cost grows with the number of agents, making efficient implementations essential for large-scale simulations. This project investigates the performance optimization of the Boids algorithm using parallel computing with OpenMP. A sequential version was implemented to establish a baseline, while a parallelized version, utilizing the Structure of Arrays (SoA) layout, was developed to exploit multicore architectures. Additionally, a comparison between Array of Structures (AoS) and SoA was performed in the sequential implementation to analyze the impact of data organization.

1.1. Understanding the Boids Algorithm

The Boids algorithm, introduced by Craig Reynolds in 1986, is a model for simulating the flocking behavior of agents such as birds, fish, or other organisms that exhibit group dynamics. Each agent, referred to as a ”boid” follows three fundamental rules:

- **Alignment**: A boid aligns its direction with the average heading of its neighbors.
- **Cohesion**: A boid moves toward the average position of its neighbors to maintain group structure.
- **Separation**: A boid steers away from nearby neighbors to avoid collisions.

(Aggiungi un immagine)

These rules are computed independently for each boid at every simulation step, creating emergent behaviors that resemble real-world flocking.

1.2. Releated Work

This project is based on the implementation described by Van Hunter Adams, which provides a concise and efficient version of the algorithm. The implementation organizes the rules into a series of updates on boid positions and velocities, iterated across all agents in the simulation.

By combining these rules, the algorithm achieves realistic collective behaviors without explicit control of the group, highlighting the power of simple local interactions in generating complex global patterns.

2. Leanguage and Libraries

The project is implemented in C++ using the OpenMP library for parallelization. OpenMP provides a straightforward way to parallelize loops and sections of code, making it suitable for the Boids algorithm, where each boid's behavior can be computed independently. The choice of C++ allows for efficient memory management and performance optimization, crucial for high-performance simulations. To visualize the simulation, the project uses the SFML (Simple and Fast Multimedia Library) for rendering the boids on the screen. SFML provides a simple interface for graphics, making it easy to create a visual representation of the simulation.

3. Methodology

3.1. Sequential Implementation

The sequential implementation of the Boids algorithm serves as the baseline for performance evaluation. In this version, the simulation iteratively updates the state of each boid by applying the three fundamental rules: alignment, cohesion, and separation. Each boid's position and velocity are recalculated at every simulation step based on the states of all other boids.

The main loop processes each boid in sequence, computing the influence of neighboring boids and updating the properties accordingly. Temporary buffers are used to store the new states before copying them back to the main data structures. This approach ensures that updates for one boid do not interfere with the calculations for others within the same iteration.

A representative pseudocode snippet of the update process is provided in the Appendix (see Section 5).

This structure is common to all implementations, with the main differences lying in how data is organized (AoS vs SoA) and whether the outer loop is parallelized. The sequential version provides a clear and straightforward reference for measuring the impact of data layout and parallelization strategies.

3.2. Data Layouts: AoS vs SoA

The project implements and compares two distinct data layouts for representing the state of the boids: Array of Structures (AoS) and Structure of Arrays (SoA).

In the **AoS** approach, each boid is represented as a structure containing all its properties (position, velocity, bias, etc.), and a vector of these structures is used to store the entire flock. This is reflected in the following structure:

```
struct BoidData {
    sf::Vector2f position;
    sf::Vector2f velocity;
    float biasval;
    int scoutGroup;
    // ...
};
std::vector<BoidData> boidDataList;
```

With this layout, the main loop iterates over the vector, and each boid's state is accessed as a whole. This approach is intuitive and convenient for object-oriented programming, but can lead to less efficient memory access patterns, especially when only a subset of fields is needed for computation.

In the **SoA** approach, each property of the boids is stored in a separate array. For example, all positions are stored in one array, all velocities in another, and so on. The structure is defined as:

```
struct BoidDataList {
    float* xPos;
    float* yPos;
    float* xVelocity;
    float* yVelocity;
    float* biasvals;
    int* scoutGroup;
    int numBoid;
    // ...
};
```

This layout enables more efficient memory access, particularly when performing operations over a single property for all boids. In the main loop, the SoA version processes each property in

bulk, which improves cache utilization and can be more easily vectorized by the compiler.

Key differences in computation:

- In the AoS version, the main loop iterates over the vector of structures, accessing and updating each boid's fields individually.
- In the SoA version, the main loop operates over arrays of properties, often using local variables to accumulate results before writing back to the arrays.
- The SoA layout allows for better spatial locality and enables optimizations such as SIMD vectorization, as memory accesses are more predictable and contiguous.

Performance advantages of SoA:

- Improved cache efficiency when accessing the same property across many boids.
- Reduced memory bandwidth usage due to contiguous memory access patterns.
- Easier application of parallelization and vectorization techniques.

A detailed example of the data structures and main loop implementations for both layouts is provided in the Appendix (see Section 5).

3.3. Parallel Implementation

The parallel implementation is based on the SoA (Structure of Arrays) version, as this layout is more suitable for parallel and vectorized operations. The main goal of the parallelization is to distribute the computation of each boid's behavior across multiple CPU threads using OpenMP.

The core idea is that the update of each boid at each simulation step is independent from the others, as all computations are based on the current state and do not interfere with updates of other boids. This makes the outer loop over boids a natural candidate for parallelization.

In the parallel version, the following changes were introduced:

- The main update loop over all boids is annotated with `#pragma omp parallel for`, allowing OpenMP to assign different iterations (boids) to different threads.
- Each thread works with its own set of local variables for the computation of new positions, velocities, and bias values, avoiding race conditions.
- Temporary arrays are allocated to store the updated properties for all boids. After the parallel loop, these arrays are copied back to the main SoA structure in a separate loop, which can also be vectorized (e.g., with `#pragma omp simd`).
- The number of threads can be configured at runtime, and OpenMP automatically manages the workload distribution.

This approach ensures that all memory accesses within the parallel region are thread-safe, as each thread writes to a unique index in the temporary arrays. The SoA layout further improves performance by enabling contiguous memory access and better cache utilization.

A simplified example of the parallelized loop is provided in the Appendix (see Section 5).

4. Results and Discussion

4.1. Performance Metrics

To evaluate the performance of the different implementations, several metrics were considered, such as execution time and speedup. This section will describe the testing environment and the methodology used to measure these metrics.

4.2. AoS vs SoA

The performance comparison between the AoS and SoA layouts will be presented in this section. It will discuss the advantages and disadvantages of each representation concerning memory access patterns and cache utilization.

4.3. Speedup Analysis

The speedup achieved through parallelization will be analyzed, comparing the execution time of the sequential and parallel versions. This section will discuss the impact of the chosen parallelization strategy and data layout on the overall performance.

5. Conclusion

This project successfully demonstrated the significant impact of parallel programming and optimized data layouts on the performance of the Boids simulation. The parallel implementation using OpenMP achieved substantial speedup compared to the sequential version. Moreover, the choice between AoS and SoA data layouts notably influenced the performance, with SoA generally providing better cache performance and memory access patterns.

Future work could explore further optimizations, such as tuning OpenMP parameters, experimenting with different chunk sizes, or exploring other parallel programming models. Additionally, investigating the impact of these optimizations on different hardware architectures would provide a broader understanding of their effectiveness.

References

Appendix

Sequential Update Loop Example

```
for (int i = 0; i < num_boids; i++) {
    for (int j = 0; j < num_boids; j++) {
        if (i != j) {
            // Compute distance and apply rules
        }
    }
    // Update velocity and position
}
```

AoS vs SoA Data Structures and Main Loop

AoS Example:

```
// Structure definition
struct BoidData {
    sf::Vector2f position;
    sf::Vector2f velocity;
    float biasval;
    int scoutGroup;
};
// Main loop (simplified)
for (int i = 0; i < boidDataList.size(); i
    BoidData refBoid = boidDataList[i];
    // ...compute and update refBoid...
    boidDataList_tmp.emplace_back(refBoid)
}
```

SoA Example:

```
// Structure definition
struct BoidDataList {
    float* xPos;
    float* yPos;
    float* xVelocity;
    float* yVelocity;
    float* biasvals;
    int* scoutGroup;
    int numBoid;
};
// Main loop (simplified)
for (int i = 0; i < N; i++) {
    float tmp_pos_x = xPos[i];
    float tmp_pos_y = yPos[i];
    // ...compute and update local variabl
    xPos[i] = new_xPos[i];
    yPos[i] = new_yPos[i];
    // ...
}
{
    {
```

Parallel SoA Main Loop Example

```
// Parallel update using OpenMP
#pragma omp parallel for schedule(static)
for (int i = 0; i < N; i++) {
    // Read current state from SoA arrays
    float tmp_pos_x = xPos[i];
    float tmp_pos_y = yPos[i];
    float tmp_vel_x = xVelocity[i];
    float tmp_vel_y = yVelocity[i];
    // ...compute new state using local va
    new_xPos[i] = ...;
```

```
        new_yPos[i] = ...;
        new_xVelocity[i] = ...;
        new_yVelocity[i] = ...;
        new_biasvals[i] = ...;
    }
    // Copy results back to main arrays (can be vectorized)
    #pragma omp simd
    for (int i = 0; i < N; i++) {
        xPos[i] = new_xPos[i];
        yPos[i] = new_yPos[i];
        // ...
    }
```