Parallelization of the Boids Algorithm

Fabio Sucameli
Department of Information Engineering
University of Florence
fabio.sucameli@edu.unifi.it

Abstract

The Boids algorithm is widely used for simulating flocking behavior in artificial agents, such as birds or fish. This project focuses on transitioning the algorithm from a sequential implementation to an efficient parallel version using OpenMP. The work explores performance optimization techniques, such as memory access patterns, workload distribution, and contention reduction, to achieve significant speedup. This report details the sequential design, the challenges of parallelization, and the final performance evaluation.

1. Introduction

The Boids algorithm, introduced by Craig Reynolds in 1986, simulates the collective motion of agents such as birds or fish. Each agent in the simulation follows simple rules: separation, alignment, and cohesion. These rules result in complex, emergent group behaviors.

In this project, we implemented the Boids algorithm in C++ and explored its optimization and parallelization. Starting from a sequential design, we applied techniques to improve its efficiency, such as better memory locality, reduction of branching, and loop-level optimizations. Finally, we parallelized the algorithm using OpenMP, achieving significant improvements in execution time.

2. Functions Overview

This section provides a brief overview of the key functions implemented in the Boids algorithm. Each function is designed to handle specific aspects of the simulation, from initialization to update logic, ensuring both correctness and performance.

2.1. Utility Functions

- **`normalize(float& x, float& y)`**: Normalizes a 2D vector to unit length if its magnitude is greater than zero. This ensures that directional biases, velocities, and other vector-based calculations are consistent.

$$\text{length} = \sqrt{x^2 + y^2}, \quad x = \frac{x}{\text{length}}, \quad y = \frac{y}{\text{length}}$$

- **`checkEdges(float& posX, float& posY, float& velX, float& velY)`**: Checks if a boid is near the edges of the simulation window. If it is, the function adjusts its velocity to keep it within bounds.

- **`enforceSpeedLimits(float& velX, float& velY)`**: Ensures that each boid's speed remains between defined minimum (`MIN_SPEED`) and maximum (`MAX_SPEED`) thresholds.

2.2. Data Initialization

- **`resizeBoidData(BoidData& boidData, size_t numBoids)`**: Resizes all vectors in the `BoidData` structure to handle a given number of boids. This function is essential for dynamic scaling of the simulation.

- **`initializeBoids(int numBoids, BoidData& boidData)`**: Initializes boid properties with random positions, velocities, and group assignments. This ensures a varied and realistic starting state for the simulation.

2.3. Core Update Logic

- **`updateBoids(BoidData& boidData)`**: Updates the positions and velocities of all boids based on rules of interaction (separation, alignment, cohesion). It uses OpenMP for parallelization to improve performance in large simulations. Key steps include:
  - Calculating average positions and velocities of neighbors within the visual range.
  - Adjusting velocities to avoid collisions and align with neighbors.

  - Applying direction bias for scout groups.

2.4. Visualization and Debugging

- **`drawBoids(sf::RenderWindow& window, const BoidData& boidData)`**: Draws boids as triangular shapes on the SFML window, with their orientation reflecting their current velocity direction.

- **`printPositions(const BoidData& boidData, const int positionsToPrint)`**: Outputs the positions of a specified number of boids to the console. Uses OpenMP to parallelize string construction and minimize contention during printing.

These two functions were excluded during execution time measurements to avoid performance impacts but remain fully functional for future debugging or other purposes.

3. Sequential Implementation

The initial implementation focused on correctness and simplicity, using a structure-of-arrays (SoA) design for better memory access patterns.

3.1. Key Features of the Sequential Algorithm

- **Structure-of-Arrays (SoA):** Positions, velocities, and other properties of agents were stored in separate vectors to improve cache performance.

- **Locality of Access:** The algorithm ensures spatial and temporal locality by iterating through the agents sequentially.

- **Collision Avoidance:** A protected range is defined to ensure agents avoid overlapping.

- **Algorithmic Improvements:** Use of temporary variables to reduce repeated memory accesses and avoid unnecessary calculations.

- **Floating-Point Reproducibility:** Ensuring consistent results by avoiding race conditions in floating-point operations.

3.2. Challenges in Sequential Design

While the sequential version was correct, it became apparent that the algorithm's execution time scaled poorly with the number of agents. This limitation motivated the need for a parallel approach.

4. Optimizing the Sequential Algorithm

Before parallelization, several optimizations were applied:

- **Branching Reduction:** Conditional checks were minimized to improve loop performance.

- **Padding for Cache Alignment:** 'alignas' was used to reduce false sharing in cache-sensitive structures.

- **Temporary Variables:** Use of local temporary variables (e.g., caching agent positions) reduced memory access overhead.

- **Avoiding Race Conditions:** Careful design ensured the algorithm would remain robust when extended to parallel execution.

5. Parallelization of the Boids Algorithm

5.1. Approach

The algorithm was parallelized using OpenMP, focusing on the computational bottleneck: the loop that updates agent positions and velocities. Key techniques include:

- **Loop-Level Parallelism:** OpenMP 'pragma omp parallel for' was applied to divide the workload among threads.

- **Static scheduling:** `(schedule(static, 32))` with fixed chunks was chosen to reduces scheduling overhead, leading to better performance in most scenarios.

- **Wall Clock Timing:** `omp_get_wtime()` was used to measure execution time for different thread counts.

5.2. Parallelization Details

- OpenMP directives such as `#pragma omp parallel for` and `#pragma omp critical` are used to parallelize computationally intensive loops and synchronize outputs, respectively.

- Temporary buffers and thread-local variables reduce contention and improve performance in multi-threaded environments.

5.3. Challenges and Solutions

- **Memory Bandwidth Bottlenecks:** Reorganizing data structures to ensure contiguous memory access. This optimization reduced memory latency and enhanced memory throughput.

- **Memory Access Patterns:** Aligning data structures and precomputing frequently accessed variables improved cache efficiency.

This modular approach to function design ensures that each component of the Boids simulation is efficient, maintainable, and scalable.

6. Performance Results

The execution times for both sequential and parallel implementations were measured, varying the number of threads and problem size. The results are shown in the tables below. They highlight the impact of parallelization, including both performance improvements and limitations due to thread contention and overhead.

6.0.1 Small Problem Size: 500 Boids and 1000 Positions

Threads	Execution Time (ms)	Positions Processed
Sequential	4.99988	1000
1	3.99995	1000
2	6.00004	1000
4	8.99982	1000
8	8.00014	1000
16	6.99997	1000
32	5.00011	1000
64	8.99982	1000
128	13.00000	1000
256	18.00010	1000

Table 1. Execution times for 500 Boids and 1000 positions.

For small problem sizes, the overhead of thread management often dominates the benefits of parallelization. Performance is better with fewer threads due to reduced contention.

6.0.2 Medium Problem Size: 5000 Boids and 10000 Positions

Threads	Execution Time (ms)	Positions Processed
Sequential	457	10000
1	447	10000
2	283	10000
4	188	10000
8	126	10000
16	119	10000
32	143	10000
64	129	10000
128	135	10000
256	188	10000

Table 2. Execution times for 5000 Boids and 10000 positions.

In medium problem sizes, the benefits of parallelization become apparent as more threads are used, achieving a significant reduction in execution time. However, performance drops slightly beyond 16 threads due to contention and thread management overhead.

6.0.3 Large Problem Size: 50000 Boids and 100000 Positions

Threads	Execution Time (ms)	Positions Processed
Sequential	50726	100000
1	46320	100000
2	26480	100000
4	16177	100000
8	12137	100000
16	18854	100000
32	21347	100000
64	20785	100000
128	20756	100000
256	22697	100000

Table 3. Execution times for 50000 Boids and 100000 positions.

For large problem sizes, parallelization yields a significant speedup up to 8 threads. Beyond this point, contention, memory access overhead, and diminishing returns from parallelism lead to reduced gains.

6.0.4 Discussion of Results

The results show that parallelization effectively reduces execution time for medium and large problem sizes. However, performance gains are limited by thread contention and overhead as the number of threads increases. For small problem sizes, the overhead of parallelism outweighs its benefits, resulting in longer execution times with more threads. These findings highlight the importance of balancing problem size and thread count to achieve optimal performance.

7. Conclusion

7.1. Performance of the Parallel Implementation

The project successfully transitioned the Boids algorithm from a sequential to a parallel implementation. The final parallel version demonstrated significant performance improvements, achieving a speedup of approximately 4.17x with 8 threads for a simulation of 50,000 Boids and 100,000 positions. These results highlight the potential of parallel computing for large-scale simulations, particularly when problem sizes are sufficient to amortize the overhead introduced by parallelism.

7.2. Hardware Constraints and Variability

The tests were conducted on my Windows PC with 4 physical cores, which likely influenced the performance and scaling results. It is important to note that execution times and speedups can vary significantly depending on hardware configurations, including factors such as core count, memory bandwidth, and cache size. The limited number of physical cores restricted the effectiveness of scaling with higher thread counts.

7.3. Challenges in Parallelization

While the parallel implementation achieved significant improvements, diminishing returns were observed when the number of threads exceeded the available physical cores. For smaller problem sizes, thread contention and overhead from memory access further reduced the gains from parallelization. These issues underline the challenges of optimizing for highly parallel workloads on limited hardware.

7.4. Future Work

Future work could explore adaptive thread management to optimize resource utilization dynamically. Additionally, improving memory access patterns and reducing synchronization overhead would further increase the algorithm's efficiency for large-scale simulations.

References

[1] Hunter Adams, *Boids Algorithm Tutorial and Implementation*, `https://vanhunteradams.com/Pico/Animal_Movement/Boids-algorithm.html`, Accessed: November 2024.

[2] Simple and Fast Multimedia Library (SFML), `https://www.sfml-dev.org/`, Accessed: November 2024.