Parallelising Boids with OpenMP

Parallel Programming for Machine Learning course project

Leonardo Bessi
E-mail address
leonardo.bessi@edu.unifi.it

Abstract

This project implements an efficient, parallel simulation of the "Boids" flocking model (Reynolds [1]) using OpenMP to achieve significant performance speedup in agent-based spatial interactions.

Future Distribution Permission

1. Introduction

Boids is an artificial life program developed by Craig Reynolds in 1986 that simulates the flocking behavior of birds and related group motion. The name "boid" corresponds to a shortened version of "bird-oid object." The key idea behind this work is that the emergent, coherent behavior of bird flocks arises from a simple observation: each bird, or boid, acts independently based on its local environment and sensory inputs.

1.1. Three Key Rules

The three fundamental rules that govern a boid's movement are:

1. **Alignment**: Steer toward the average heading of local neighbors;
2. **Cohesion**: Steer toward the "center of mass" (average position) of local neighbors;
3. **Separation**: Steer to avoid crowding or colliding with very close neighbors.

The *neighbors* are defined as all other boids located within a specific perception radius r.

1.2. The Algorithm

The simulation operates on discrete time steps. During each iteration, every agent undergoes a two-phase update process:

1. **Perception and Force Calculation**: Each boid scans the environment to identify neighbors within its perception radius. It then calculates three steering vectors based on the rules of alignment, cohesion, and separation. These vectors are weighted and summed to produce a final acceleration vector.
2. **State Integration**: The acceleration is applied to the current velocity, which is then clamped to a maximum speed to maintain stability. Finally, the velocity is added to the current position to move the agent.

In the naive implementation, identifying neighbours for a single boid requires iterating through all other $N-1$ boids to check for distances. Since this check must be performed for every agent in every frame, the total complexity per frame is $O(N^2)$, making the simulation practically unscalable as the number of agents increases.

2. Implementation

The implementation follows Shiffman [2].

2.1. Spatial Partitioning

The simulation environment is defined by a fixed height and width. To optimize neighbor lookups, the world is partitioned into a uniform grid where each square cell's side length equals

Table 1. Comprehensive performance analysis across Problem Sizes

Scenario (N)	Threads	Avg (s)	95th %ile (s)	99th %ile (s)	Speedup
1,000 Boids	1 (Seq)	0.0005	0.0006	0.0008	1.00×
	24	0.0003	0.0003	0.0004	1.70×
10,000 Boids	1 (Seq)	0.0118	0.0180	0.0226	1.00×
	24	0.0026	0.0029	0.0029	4.54×
50,000 Boids	1 (Seq)	0.2215	0.3475	0.4233	1.00×
	12	0.0176	0.0225	0.0251	12.54×
	24	0.0135	0.0186	0.0244	16.43×

the boids' perception radius. The grid is implemented as a 2D matrix of `GridCell` structures, each containing an array of `Boid` pointers and a counter for the current population size.

At the beginning of each frame, the grid is cleared. The population is then iterated through, and each boid pointer is assigned to its corresponding cell based on its spatial coordinates. To find the neighbors of a specific boid, the algorithm only inspects the boid's current cell and the 8 surrounding adjacent cells. This spatial partitioning significantly reduces the number of distance checks compared to a brute-force approach.

2.2. Parallelisation with OpenMP

To avoid race conditions, the simulation is split into distinct steps:

1. **Grid Clearing (Embarrassingly Parallel)**: Each thread resets a subset of the grid cells by setting their size to zero.
2. **Grid Population (Sequential)**: Each boid is assigned to its corresponding grid cell. While this could be parallelized using atomic operations or thread-local grids, it remains sequential to avoid locking overhead, as the operation's cost is negligible compared to force calculations.
3. **Force Calculation (Parallel with `schedule(dynamic)`)**: Threads calculate the steering forces for the agents. A dynamic schedule is employed to address *load imbalance* caused by the uneven spatial distribution of boids across the grid.
4. **State Update (Embarrassingly Parallel)**: Boids update their velocity and position based on the previously computed forces.

3. Performance Analysis

3.1. Test Platform

The benchmarks were conducted on a system with the following specifications:

- **CPU**: AMD Ryzen™AI 9 HX370 (12 Cores / 24 Threads)
- **RAM**: 32GB LPDDR5x @ 8000 MHz
- **C Compiler**: GNU `gcc` 15.2.1

3.2. Test Cases and Results

To evaluate the scalability of the OpenMP implementation, several scenarios were tested varying the agent count (N), the number of timesteps, and the thread count (T):

- $N = 1,000$, 1,000 timesteps, 24 threads
- $N = 10,000$, 2,000 timesteps, 24 threads (see Figure 1)
- $N = 50,000$, 1,000 timesteps, 24 threads
- $N = 50,000$, 1,000 timesteps, 12 threads (see Figure 2)

All performance metrics represent the mean of three independent execution runs using different initialization seeds. Table 1 provides a detailed performance comparison.

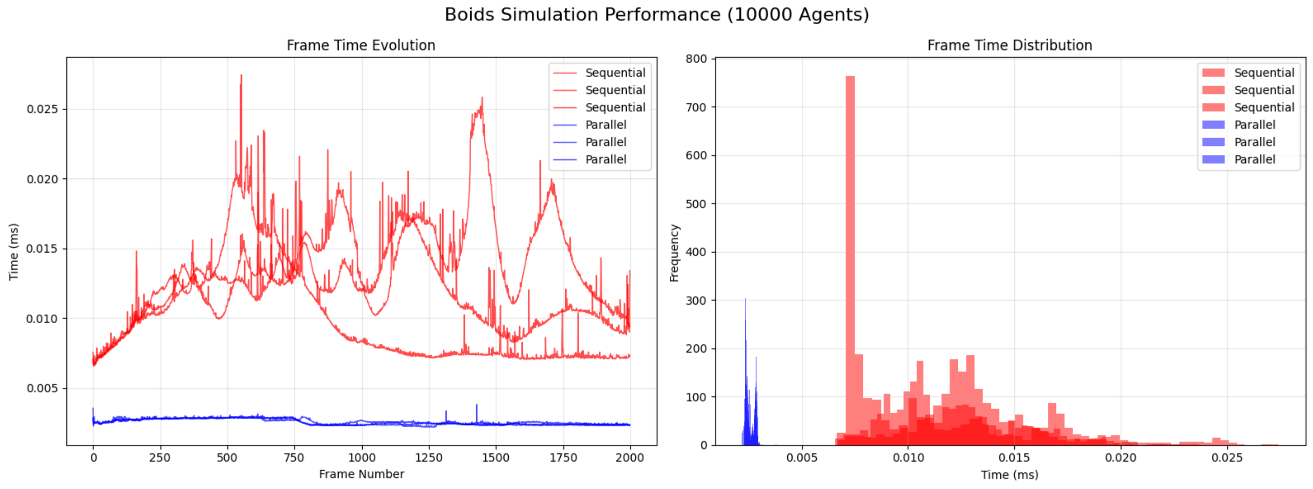

Figure 1. 10000 agents simulation on 24 OMP threads

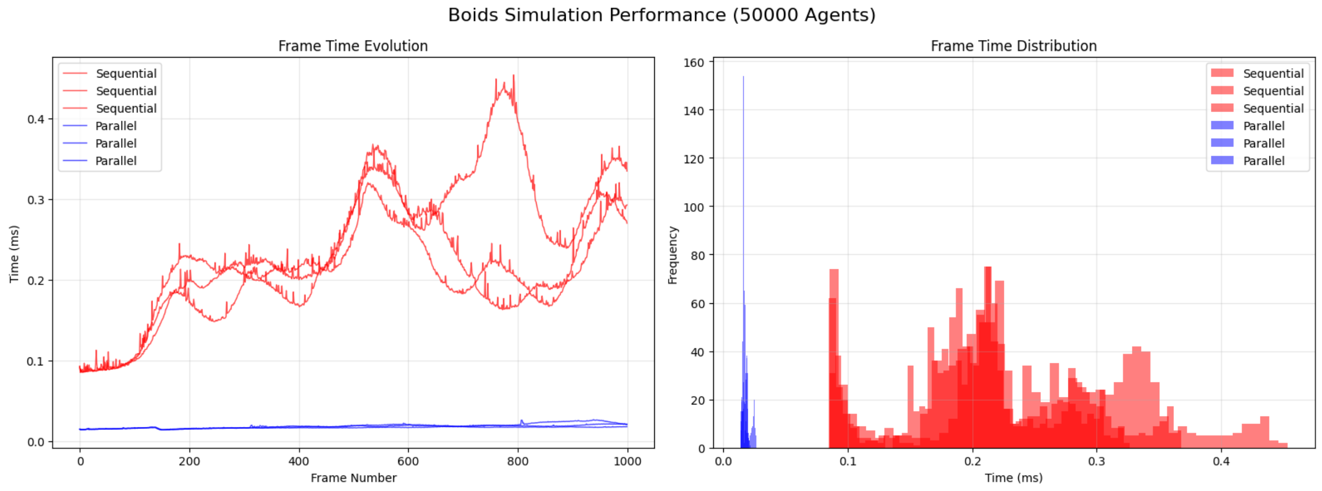

Figure 2. 50000 agents simulation on 12 OMP threads

4. Discussion of Results

The data shown in Table **??** demonstrates a significant reduction in execution time through the use of OpenMP. The 12-thread version achieves a speedup of $12.54\times$ on average. This is likely due to hotter L1 and L2 caches per core, leading to a super-linear speedup on average.

The transition from 12 to 24 cores shows diminishing returns, with the speedup increasing only to $16.39\times$. This behavior is consistent with the architecture of the Ryzen AI 9 HX370; while the CPU supports 24 threads via Simultaneous Multithreading (SMT), these logical threads share execution resources within the 12 physical cores.

The close proximity of the 95th percentile times to the average suggests that the `schedule(dynamic)` clause successfully mitigated potential load imbalance caused by localized boid clustering.

References

[1] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, Aug. 1987.

[2] D. Shiffman. *The Nature of Code: Simulating Natural Systems with JavaScript*. No Starch Press, 2024.