Simulating Flocking Behavior with Boids: Sequential, Parallel, and Distributed Approaches

Emilija Trajkovska
UP FAMNIT, Slovenia
89221090@student.upr.si

Abstract

This paper presents a three-dimensional Boids simulation rendered in two dimensions, illustrating how simple local rules—cohesion, separation, and alignment—produce emergent flocking behavior. We describe our modular implementation of each rule, detail an interactive GUI for pre-simulation setup and live slider-based tuning, and compare performance across sequential, multithreaded parallel, and MPI-distributed executions. Benchmarks over flock sizes from 50 to 2000 boids reveal trade-offs in scalability and responsiveness.

1 Introduction

Craig Reynolds' seminal Boids model (1987) demonstrated that complex group behaviors, such as flocking, can emerge from decentralized agents following just three simple rules:

- **Cohesion**: Boids are steered toward the average position of their local neighbors, encouraging the group to move as a cohesive unit.
- **Separation**: Boids maintain a minimum distance from nearby flockmates to avoid crowding and collisions.
- **Alignment**: Boids adjust their velocity to match the average direction and speed of their neighbors, promoting synchronized movement.

In our implementation, each boid navigates a three-dimensional space but is drawn on a two-dimensional canvas, where its size encode depth. At every time step, a boid computes three steering vectors—one for cohesion, one for separation, and one for alignment—based solely on its local neighborhood, combines them according to user-defined weights, and updates its velocity and position accordingly. This decentralization not only yields lifelike flocking but also lends itself to parallel and distributed computing strategies.

2 Graphical Interface and Parameter Control

Our simulation GUI consists of a setup screen followed by the main simulation view. On the setup screen, the user specifies the world width and height, initial boid speed and the number of boids. Upon launching, the simulation screen renders the flock in real time and displays three horizontal sliders for adjusting cohesion, separation, and alignment weights. Manipulating these sliders immediately alters the steering forces: higher cohesion weights pull boids into tighter clusters, increased separation encourages them to disperse, and elevated alignment leads to more synchronized motion. (Figure 1).

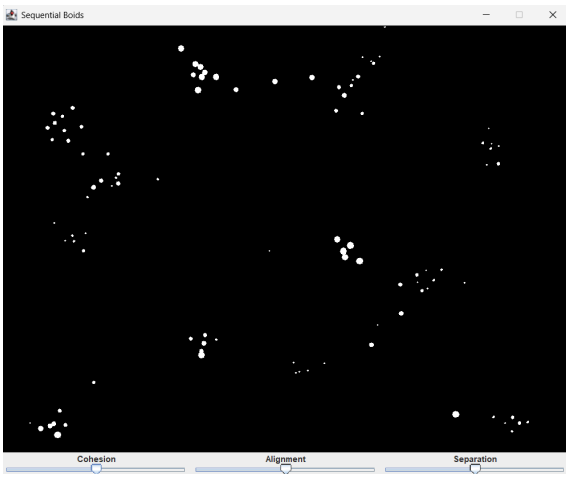

Figure 1: Real-time simulation view with sliders for cohesion, separation, and alignment. Boid size indicates depth in 3D space.

3 Flocking Rules

Each rule yields a steering vector that influences the boid's acceleration. These vectors are weighted by the current slider values and summed. The resulting vector is limited to a maximum force to ensure smooth motion.

The **cohesion** rule steers a boid toward the centroid of its neighbors within a radius R_c. By averaging neighbor positions, subtracting the boid's current position, normalizing, scaling by the maximum speed, and then subtracting the current velocity, we obtain a gentle pull toward the group's center without abrupt turns:

```
Vector3D cohesion(ArrayList<Boid> boids) {
    Vector3D steering = new Vector3D(0, 0, 0);
    int total = 0;
    for (Boid other : boids) {
        double d = position.distance(other.position);
        if (other != this && d < perceptionRadius) {
            steering.add(other.position);
            total++;
        }
    }
    if (total > 0) {
        steering.divide(total);
        steering.subtract(position);
        steering.normalize();
        steering.multiply(maxSpeed);
        steering.subtract(velocity);
        steering.limit(maxForce);
    }
    return steering;
}
```

The **separation** rule prevents overcrowding by generating a repulsive force inversely proportional to the square of the distance for each neighbor closer than R_s. Summing and normalizing these repulsions yields a smooth avoidance maneuver:

```
Vector3D separation(ArrayList<Boid> boids) {
    Vector3D steering = new Vector3D(0, 0, 0);
    int total = 0;
    for (Boid other : boids) {
        double d = position.distance(other.position);
        if (other != this && d < perceptionRadius) {
            Vector3D diff = Vector3D.subtract(position, other.position);
            diff.divide(d * d);  // Stronger repulsion for closer boids
            steering.add(diff);
            total++;
        }
    }
    if (total > 0) {
        steering.divide(total);
        steering.normalize();
        steering.multiply(maxSpeed);
        steering.subtract(velocity);
        steering.limit(maxForce);
    }
    return steering;
}
```

Under the **alignment** rule, a boid matches the average heading of neighbors within R_a. By summing neighbor velocities, normalizing, scaling, and subtracting the boid's current velocity, the rule fosters cohesive directional alignment:

```
Vector3D alignment(ArrayList<Boid> boids) {
    Vector3D steering = new Vector3D(0, 0, 0);
    int total = 0;
    for (Boid other : boids) {
        double d = position.distance(other.position);
        if (other != this && d < perceptionRadius) {
            steering.add(other.velocity);
            total++;
        }
    }
    if (total > 0) {
        steering.divide(total);
        steering.normalize();
        steering.multiply(maxSpeed);
        steering.subtract(velocity);
        steering.limit(maxForce);
    }
    return steering;
}
```

These modular implementations allow easy adjustment or replacement of individual behaviors without affecting the overall simulation structure.

4 Execution Architectures

In **sequential** mode, a single loop updates all boids each frame, collecting neighbors and computing forces one boid at a time. While simple, its

$O(n^2)$ complexity limits real-time performance for large flocks.

The **parallel** mode partitions the boid list into chunks processed concurrently by worker threads. Shared slider values are read once per frame, and thread-safe buffers collect updated states.

For the **distributed** mode, MPJ Express orchestrates a master–worker pattern. The master process broadcasts current slider values and partitions boids among workers. After computing updates locally, each worker sends back new positions and velocities. Though this enables scaling beyond a single machine's resources, frequent MPI messages introduce latencies that reduce benefit for moderate-scale simulations.

5 Performance Evaluation

We tested all modes by measuring the average time to process 200 frames across flock sizes of 50, 100, 500, 1000, and 2000 boids. The sequential and parallel implementations were tested on an 8-core workstation, while the distributed version was evaluated using 4 MPI processes.

Table 1 presents the results:

Table 1: Execution Times (s) for 200 Frames

Boids	50	100	500	1000	2000
Sequential	0.19	0.56	5.25	12.52	50.18
Parallel	0.03	0.20	2.84	7.75	25.71
Distributed	0.10	0.75	6.19	13.55	33.97

Figure 2 visualizes these trends, highlighting the crossover point where distributed execution begins to outperform sequential processing.

The testing results demonstrate that the optimal implementation strategy depends on flock size - the sequential version performs best for small flocks ($<$ 100 boids) due to minimal overhead, while the parallel approach achieves 2-3$\times$ speedups for medium-to-large flocks (100-2000 boids) by efficiently distributing work across CPU cores. The distributed MPI implementation, though theoretically scalable, incurs significant communication latency that outweighs its benefits for this real-time simulation, making it less practical than the multithreaded solution except potentially for extremely large flocks (>2000 boids) distributed across multiple machines. These findings confirm that parallelization via multithreading offers the best balance of performance and simplicity for interactive boid simulations running on a single multicore system.

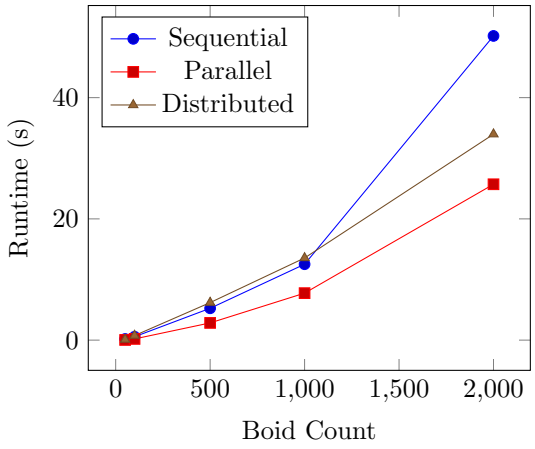

Figure 2: Performance scaling across execution modes.

6 Conclusion

Our Boids simulation demonstrates how local interaction rules yield emergent flocking behavior and how different computational architectures affect its performance. Multithreaded execution offers the best real-time responsiveness on multicore hardware, while MPI-based distribution enables very large-scale simulations at the cost of communication latency. The integrated GUI and sliders facilitate rapid experimentation with behavioral parameters, making this system a versatile platform for both research and interactive demonstration.

References

[1] C. W. Reynolds, "Flocks, Herds and Schools: A Distributed Behavioral Model," SIGGRAPH, 1987.

[2] E. Trajkovska, "Boids Simulation Repository," GitHub, 2025. `https://github.com/EmilijaTR/Boids`

[3] D. Shiffman, "The Nature of Code: Simulating Natural Systems with Processing," 2012.