

# Rabbits and Foxes Ecosystem Simulation

**Project done by:** Diogo Alves, 202006033 and Mário Minhava, 202206190

## Project Overview

This project implements a parallel solution for simulating a **Rabbits and Foxes Ecosystem** using **multi-threading with pthreads**. The implementation leverages parallel computing to distribute the simulation workload across multiple threads, enabling efficient processing of large ecosystem grids while maintaining ecosystem dynamics and inter-entity interactions.

## Algorithm Implementation

### Ecosystem Simulation Problem

The algorithm simulates a predator-prey ecosystem where rabbits and foxes interact according to specific behavioral rules. The simulation progresses through discrete generations, with each generation consisting of:

1. **Rabbit Movement Phase:** All rabbits attempt to move to adjacent empty cells
2. **Rabbit Conflict Resolution:** Handle rabbit movements that cross thread boundaries
3. **Fox Movement Phase:** Foxes move toward prey or empty cells
4. **Fox Conflict Resolution:** Handle fox movements and finalize generation state

### Entity Behavior Rules

#### Rabbit Behavior:

- Move to adjacent empty cells (north, east, south, west)
- Cannot move diagonally or into rocks/occupied cells
- Reproduce after reaching procreation age
- Die of old age or when eaten by foxes

#### Fox Behavior:

- Prefer moving toward cells containing rabbits
- Can move to empty cells if no prey available
- Require regular feeding to survive (food age tracking)
- Reproduce after reaching procreation age
- Die from starvation or old age

**Conflict Resolution:** When multiple entities attempt to move to the same cell, resolution follows priority rules:

- **Rabbits:** Entity closest to procreation survives

- **Foxes:** Entity closest to procreation survives; if tied, entity furthest from starvation survives

### Parallel Algorithm Approach

The parallel implementation divides the ecosystem grid by rows, with each thread responsible for a specific region. This approach provides:

1. **Dynamic Load Balancing:** Thread regions are assigned based on entity density rather than fixed grid divisions
2. **Minimal Synchronization:** Only adjacent threads need to communicate for boundary movements
3. **Conflict Management:** Cross-boundary movements are handled through structured conflict resolution

### Project Structure

The project follows a modular architecture with clear separation of concerns:

File	Purpose
<code>main.c</code>	Program entry point, command-line argument processing
<code>rabbitsandfoxes.h/.c</code>	Core simulation logic, generation processing
<code>entities.h/.c</code>	Entity creation, destruction, and lifecycle management
<code>movements.h/.c</code>	Movement analysis and conflict detection
<code>threads.h/.c</code>	Thread management, synchronization, and parallel control
<code>matrix_utils.h/.c</code>	Grid utilities and memory management
<code>output.h/.c</code>	Ecosystem state display and result formatting
<code>makefile</code>	Build system with compilation and comprehensive testing

### Architecture and Data Structures

To ensure memory efficiency and report conciseness, the core data structures are defined below, linking conceptual components to their specific struct implementations.

#### 1. Grid Cell Representation (`WorldSlot`)

The `WorldSlot` struct is optimized for size, representing a single coordinate in the ecosystem.

Component	Struct Member	Purpose
Type	<code>slotContent</code>	Identifies content: <code>Empty</code> , <code>Rock</code> , <code>Rabbit</code> , or <code>Fox</code> .

Component Struct Member	Purpose
<b>Move Cache</b>	Pre-computed valid directions (N, S, E, W) to reduce runtime checks.
<b>Entity Data</b>	Memory-efficient storage. Holds <code>age</code> for Rabbits, or <code>age + hunger</code> for Foxes.

## 2. Thread Synchronization (`ThreadedData`)

The `ThreadedData` struct manages the parallel execution context and inter-thread communication.

Primitive	Struct Member	Usage
<b>Barriers</b>	<code>barrier</code>	Ensures all threads complete a movement phase before starting the next.
<b>Semaphores</b>	<code>threadSemaphores</code>	Controls access to boundary rows between adjacent threads.
<b>Buffers</b>	<code>conflictPerThreads</code>	Temporary storage for entities attempting to move across thread boundaries.

## Core Functions and Algorithm Logic

### 1. Parallel Simulation Architecture

#### Main Coordination Functions:

Function	Responsibility
<code>runParallelSimulation()</code>	Thread creation, workload distribution, timing measurement
<code>executeParallelGeneration()</code>	Single generation execution across all threads
<code>executeWorkerThread()</code>	Individual thread execution loop

## 2. Dynamic Work Distribution Strategy

**Entity-Density Load Balancing:** The system assigns work based on entity density rather than fixed row divisions. Binary search finds optimal row boundaries targeting equal entities per thread.

## 3. Generation Processing Pipeline

### Two-Phase Execution:

1. **Rabbit Phase:** Threads create local snapshots, process rabbit movements, defer cross-boundary conflicts, synchronize
2. **Fox Phase:** Update snapshots, process fox movements (prioritizing prey), resolve conflicts, update entity counts

Deterministic movement uses `(generation + row + col) % possibleMoves` ensuring reproducible results across thread counts.

## 4. Entity Behavioral Models

**Rabbits:** Move to empty adjacent cells. Reproduce when reaching maturity. Age-based conflict resolution (older wins).

**Foxes:** Prioritize moves toward prey, fallback to empty cells. Increment starvation counter each turn, die if exceeding threshold. Age-based conflicts with food level tiebreaker.

## 5. Inter-Thread Coordination

### Synchronization Strategy:

- **Global Barriers:** All threads synchronize between rabbit and fox phases
- **Local Semaphores:** Adjacent threads coordinate boundary conflict resolution
- **Conflict Buffers:** Thread-safe storage for cross-boundary movements

**Communication Pattern:** Threads only interact with immediate neighbors, reducing complexity from  $O(n^2)$  to  $O(n)$ .

## Technical Implementation Details

**Memory Management:** Dynamic entity allocation with union-based storage. Pre-computed movement directions cached per position. Contiguous world matrix allocation for cache efficiency.

**Thread Safety:** Territorial ownership (threads own specific rows) + ordered synchronization prevents race conditions. Barrier synchronization eliminates deadlocks. Conflict buffers isolate cross-boundary operations.

## Performance Results

### Execution Times and Speedup Analysis

Input Size	Sequential	2 Threads	4 Threads	8 Threads	16 Threads
<b>5x5</b>	0.001s	0.000s (1.80x)	0.000s (1.52x)	-	-
<b>10x10</b>	0.001s	0.004s (0.29x)	0.005s (0.27x)	0.008s (0.15x)	-
<b>20x20</b>	0.014s	0.049s (0.28x)	0.044s (0.32x)	0.081s (0.17x)	0.157s (0.09x)
<b>100x100</b>	4.148s	2.606s (1.59x)	1.578s (2.63x)	1.501s (2.76x)	2.033s (2.04x)
<b>100x100_unbal01</b>	2.818s	2.002s (1.41x)	1.268s (2.22x)	1.313s (2.15x)	1.915s (1.47x)
<b>100x100_unbal02</b>	3.951s	2.665s (1.48x)	1.577s (2.50x)	1.512s (2.61x)	2.016s (1.96x)
<b>200x200</b>	16.622s	9.236s (1.80x)	5.106s (3.26x)	3.489s (4.76x)	3.408s (4.88x)

### Performance Analysis Discussion

**Key Results:** - **Optimal Performance:** 4-8 threads achieve 2.5-4.7x speedup on 100x100+ ecosystems - **Thread Overhead:** Small ecosystems ( $<=20\times20$ ) show negative speedups due to synchronization costs exceeding computation - **Scalability Limit:** 16+ threads cause performance regression from increased barrier synchronization and conflict resolution overhead - **Load Balancing:** Dynamic allocation effectively handles unbalanced inputs, maintaining similar performance across test variants

NOTE: In the project folder there are 2 plots showcasing some plots for more info path: benchmark\_plots

### Main Difficulties

**Cross-Boundary Movement Coordination:** The most challenging aspect was handling entity movements across thread boundaries without introducing race conditions. The solution required implementing a deferred conflict resolution system where movements are registered but not executed immediately, followed by synchronized resolution phases.

**Dynamic Load Balancing Implementation:** Achieving effective load distribution based on entity density rather than fixed grid partitions required some care. The challenge was maintaining efficiency while preventing workload imbalances as entity populations shifted during simulation.

**Deterministic Parallel Execution:** Ensuring identical results across different thread counts demanded careful synchronization point placement and deterministic movement selection algorithms.

**Performance Optimization vs. Correctness:** Balancing synchronization overhead with computational parallelism proved difficult. Too few synchronization points introduced race conditions, while excessive synchronization eliminated performance gains.

## Conclusion

This project implements a parallel rabbits and foxes ecosystem simulation demonstrating advanced concurrent programming techniques. The modular architecture separates concerns effectively, with dedicated modules for entity management, movement analysis, thread coordination, and conflict resolution. The implementation showcases sophisticated parallel algorithms including dynamic load balancing, territorial thread ownership, and structured synchronization patterns that ensure deterministic results across different thread configurations while maintaining thread safety.