Project Assignment II: Ecosystem Simulation

Kathleen Soares
up201903010

Sérgio Cardoso
up202107918

December 10, 2025

1 Introduction

This report describes the implementation and evaluation of a parallel ecosystem simulator developed for this project. The objective was to model interactions between rabbits, foxes and rocks on a 2D grid and to analyse the effect of introducing OpenMP-based parallelism. The document provides a concise overview of the algorithm in question, the adopted parallelisation strategy, performance results, and the main difficulties encountered during development.

2 Algorithm Overview

The simulator operates on an $R \times C$ grid where each cell may contain a rabbit, fox, rock, or be empty. The simulation evolves for N_GEN generations, each consisting of two phases executed sequentially:

- **Rabbit:** move to adjacent empty cells. Reproduction occurs after a fixed number of generations (`GEN_PROC_RABBITS`) and rabbits left one child in the cell they vacated. After reproduction and leaving cell, the age of counter resets.
- **Fox:** move preferentially towards adjacent rabbits; otherwise they move to empty cells. They reproduce after `GEN_PROC_FOXES` generations and die if they do not eat for `GEN_FOOD_FOXES` generations. Foxes also leave one child in the cell they vacated upon reproduction, resetting their age counter.

Two grids (`grid1` and `grid2`) are used in a double-buffering scheme. Each cell stores the entity type and age counters. Conflict resolution uses a simple priority rule: older animals, or (for foxes) animals with lower food age, overwrite younger ones.

Conflicts resolution rules

When multiple animals attempt to move into the same cell, the following rules are applied:

- **Rabbits:** the rabbit with the highest procreation age (`PROC_AGE`) prevails. Younger rabbits lose the conflict and are not placed in the target cell.
- **Foxes:** if multiple foxes attempt to occupy the same cell, the fox with the highest procreation age prevails. In case of a tie, the fox with the lowest food age (`FOOD_AGE`) is selected, reflecting preferential survival of less-starved individuals.

In each generation, every animal increases its procreation age (`PROC_AGE`) by one unit. For foxes, the food age (`FOOD_AGE`) also increases by one when they do not eat during that generation.

3 Parallel Implementation

Parallelisation was introduced using OpenMP. A single parallel region encloses the full simulation, and parallel work is expressed via `#pragma omp for`. Each generation includes:

- Initialisation of the output grid.
- Processing of all rabbits or foxes using `schedule(guided)`.

3.1 Implementation Details and Functions

- `init_grids`: Initialises the two grids and allocates memory for the lock table.
- `destroy_grids`: Frees the allocated memory for the grids and lock table.
- `get_adjacent_index`: Retrieves the indices of adjacent cells for a given cell, considering grid boundaries.
- `solve_rabbit_conflict`: Resolves conflicts between rabbits in the same cell.
- `solve_fox_conflict`: Does pretty much the same as the previous one, but for foxes.

3.2 Synchronization

Multiple animals may attempt to move into the same cell. To ensure correctness, the parallel version uses a lock table of 65 536 OpenMP locks, mapped to grid indices via a bit mask. Each potentially conflicting update to the output grid acquires the corresponding lock, applies the conflict-resolution rules, and then releases the lock. Although correct, this introduces substantial synchronization overhead.

3.3 Load Balancing

The cost per cell depends on the presence of animals and their movement possibilities. To mitigate imbalance, `schedule(guided)` is used, providing dynamic chunk allocation with low overhead. Despite this, the inherent serialization between the rabbit and fox phases, together with fine-grained locking, limits scalability.

4 Performance Evaluation

4.1 Execution Times and Speedups

This section presents the execution times (in milliseconds) and the speedups obtained for the sequential version and for the parallel implementation using 1, 2, 4, 8, and 16 workers.

The average sequential execution time measured was $T_{\text{seq}} = 2767.548$ ms. Parallel execution times for each worker configuration were also collected and used to compute two types of speedup:

- **Sequential speedup:**
$$S_{\text{seq}}(p) = \frac{T_{\text{seq}}}{T_p}$$
which compares each parallel configuration with the sequential execution.
- **Relative speedup matrix:**
$$S_i(p) = \frac{T_i}{T_p}$$
which compares every worker configuration p against a base configuration i.

Base Workers i	**1**	**2**	**4**	**8**	**16**
1	1.000	1.410	2.227	3.192	3.558
2	0.708	1.000	1.578	2.262	2.522
4	0.448	0.633	1.000	1.433	1.597
8	0.313	0.441	0.697	1.000	1.114
16	0.280	0.396	0.625	0.896	1.000

Table 1: Speedup matrix comparing execution times between all worker configurations.

Workers p	**Execution Time** T_p **(ms)**	**Speedup** $S_{\textbf{seq}}(p)$
1	4781.816	0.578
2	3389.943	0.816
4	2146.978	1.289
8	1498.052	1.847
16	1343.618	2.059

Table 2: Speedup in relation to the sequential execution time.

Table 1 shows that the parallel version exhibits internal scalability: configurations with more workers outperform those with fewer workers. However, when compared with the sequential runtime (T_{seq}), all parallel configurations remain slower, as shown in Table 2. This indicates that the overhead introduced by parallelisation dominates the useful computation.

Although the parallel implementation becomes faster as more threads are added, it never surpasses the sequential version. This occurs because parallel

execution introduces substantial overhead — including lock contention, memory-coherence traffic, and synchronization barriers — which outweighs the useful work performed.

4.2 Analysis of Results

The main causes of this lack of scalability are:

- **High synchronization overhead:** The fine-grained lock table serializes a large portion of the updates, causing threads to wait frequently.
- **Memory contention:** Both grids are accessed intensively by all workers, leading to cache invalidations and increased memory-coherence traffic.
- **Low computational granularity:** Each update performs only a small amount of useful work, while lock acquisition and release dominate the total cost.
- **Phase dependency:** The strict ordering between the rabbit and fox phases prevents additional parallelism and limits speedup.

Even though the relative speedup matrix shows that using more workers can improve performance when compared with fewer workers, the **absolute performance remains worse than the sequential version for all thread counts**.

4.3 Comparison with Previous Measurements

Workers p	**local** $S_{\mathbf{seq}}(p)$	**hyrax64** $S_{\mathbf{seq}}(p)$
1	0.553	0.578
2	0.794	0.816
4	0.911	1.289
8	0.633	1.847
16	0.416	2.059

Table 3: Comparison between local machine and hyrax64 sequential speedup values.

To contextualize the updated results obtained on `hyrax64`, Table 3 compares the new speedup values with the previous measurements taken locally. The earlier results showed performance degradation as the number of workers increased, while the updated results exhibit clear internal scalability and noticeably higher speedups.

Overall, the comparison indicates that the previous execution environment suffered from severe contention, higher synchronization costs, and poor scaling, while the `hyrax64` environment provides substantially better thread utilization.

The differences between the two sets of results arise from variations in system architecture and runtime conditions. The earlier machine exhibited stronger effects from lock contention, memory bandwidth limitations, and cache interference, which severely hindered scalability. In contrast, `hyrax64` offers more

efficient parallel execution, improved cache performance, and lower contention, allowing the algorithm to achieve significantly better speedups and more consistent internal scalability.

5 Difficulties and Comments

The main difficulty was ensuring the correctness of the ecosystem rules. Small inaccuracies in movement or reproduction generated noticeable differences, requiring iterative debugging.

The parallel version introduced the most challenges. Frequent concurrent accesses to the grid created write conflicts, making lock-based synchronization necessary. However, the resulting overhead dominated execution time and prevented speedup. Understanding these effects and tuning the implementation were important parts of the development process.

The project illustrated that parallelisation is not always beneficial, especially for algorithms with fine-grained operations and extensive shared-memory interaction. Future work could explore alternative decompositions or different parallel programming models with lower contention.

6 Conclusion

Although multiple thread configurations were tested (1, 2, 4, 8, and 16 workers), none of them outperformed the sequential execution. The highest speedup obtained relative to the sequential baseline was $S_{\text{seq}}(16) = 2.059$, which remains below 1 and confirms that the overhead introduced by parallelisation outweighs the useful computational work.

The speedup matrix shows that increasing the number of workers provides modest internal improvements (e.g., $S_8(16) = 1.114$). However, these gains are not sufficient to overcome synchronization costs, memory contention, and the fine-grained nature of the updates. This demonstrates that, for algorithms dominated by shared-memory conflicts and frequent synchronization, OpenMP parallelisation may hinder performance rather than enhance it.

References

[1] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.

[2] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald and R. Menon, *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[3] OpenMP Architecture Review Board, *OpenMP Application Programming Interface*. Version 5.2, 2023. Available at: `https://www.openmp.org/specifications/`

Note: The complete source code (`ecosystem.c` and `ecosystem_seq.c`) is attached in the ZIP file submitted along with this report.