Project Assignment II: Ecosystem Simulation

Kathleen Soares
up201903010

Sérgio Cardoso
up202107918

December 8, 2025

1 Introduction

This report describes the implementation and evaluation of a parallel ecosystem simulator developed for Project II of the Parallel Computing course. 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, 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`).
- **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.

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.

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 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 write to the output grid acquires the corresponding lock, updates the cell using the conflict-resolution rules, and releases the lock. Although correct, this introduces substantial synchronization overhead.

3.2 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 $T_{\text{seq}} = 3679.024$ 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. This provides a more complete understanding of how each configuration scales relative to all others.

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

Base Workers i	**1**	**2**	**4**	**8**	**16**
1	1.000	1.435	1.647	1.144	0.752
2	0.696	1.000	1.147	0.797	0.524
4	0.606	0.871	1.000	0.694	0.456
8	0.874	1.254	1.440	1.000	0.657
16	1.328	1.907	2.189	1.520	1.000

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

Workers p	**Execution Time** T_p **(ms)**	**Speedup** $S_{\mathbf{seq}}(p)$
1	6647.329	0.553
2	4630.680	0.794
4	4034.203	0.911
8	5809.968	0.633
16	8833.709	0.416

Table 2 shows the execution times for the sequential and parallel versions. The sequential implementation achieved the lowest time ($T_{\text{seq}} = 3679.024$ ms), while all parallel configurations resulted in longer runtimes, yielding $S_{\text{seq}}(p) < 1$ for every p.

To complement this, Table 1 presents the relative speedup matrix $S_i(p)$, comparing each worker configuration against all others. The matrix confirms that configurations with more workers do not scale and, in many cases, perform worse than those with fewer workers.

The main causes of this poor scalability are:

- **High synchronization overhead:** the fine-grained lock table serializes many updates.
- **Memory contention:** both grids are heavily accessed, creating high coherence and synchronization costs.
- **Low computational granularity:** the useful work per cell is small compared to the cost of acquiring and releasing locks.
- **Phase dependency:** fox processing depends on the rabbit phase, reducing available parallelism.

Overall, the overhead introduced by OpenMP exceeds the useful work required by the simulation, explaining why the parallel version becomes slower as the number of workers increases.

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

The project provided practical insight into parallel programming and highlighted the gap between theoretical speedup and real performance. Although a correct OpenMP parallelisation was implemented, the simulation did not benefit from multiple workers due to synchronization costs, memory contention, and phase dependencies.

Despite the lack of performance gains, the work was valuable for understanding how algorithm structure, data dependencies, and access patterns influence scalability. These lessons are essential for designing efficient parallel applications.

References

[1] P. Pacheco, *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

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

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

[4] 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.