Project Assignment II: Ecosystem Simulation

Diogo Gomes de Araújo (up202510267) Rodrigo Gomes de Araújo (up202205515)

December 10, 2025

Contents

1 Introduction

The following report was written as the second assignment in the Parallel Computing course at Faculdade de Ciências da Universidade do Porto.

2 State of the Art

2.1 Ecosystem Simulation

The problem presented consists of a deterministic ecosystem simulation similar to *Conway's Game of Life* [1], where there are three species: foxes, rabbits and rocks.

The simulation takes place in a $R \times C$ matrix where each cell represents a fox, a rabbit, a rock, or an empty space to which the living cells (foxes and rabbits) can move. It is initialized with a given population that evolves after each generation according to a predetermined set of rules. Given identical initial conditions and parameters, the system always produces the same sequence of states, making it reproducible and testable.

2.1.1 Rules for Rabbits

- Rabbits can only move horizontally or vertically to an adjacent cell.
- In each generation, rabbits attempt to move to an empty adjacent cell. If there are many, they choose one according to the common criteria on 2.1.4. If there is no empty adjacent cell, they stay in the same place.
- Rabbits can procreate whenever `GEN_PROC_RABBITS` generations have passed since they were born or since they last procreated. Whenever a rabbit reaches the age to procreate and makes a move, a new rabbit is left in its former position. Both rabbits' procreation age resets to zero, ensuring parent and offspring follow the same reproduction cycle.

2.1.2 Rules for Foxes

- Foxes can only move horizontally or vertically to an adjacent cell.
- In each generation, foxes prioritize eating rabbits by moving to an adjacent cell occupied by one. If there are many, they choose the cell according to the common criteria on 2.1.4. If no rabbit is adjacent, foxes attempt to move to an empty cell using the same criteria. If neither option exists, they stay in place.
- Foxes starve and die whenever `GEN_FOOD_FOXES` generations have passed since they were born or last ate a rabbit. They die after searching and not finding a rabbit to eat, creating a natural predator-prey balance.

- Foxes can procreate whenever `GEN_PROC_FOXES` generations have passed since they were born or since they last procreated. Whenever a fox reaches the age to procreate and makes a move, a new fox is left in its former position. Both foxes' procreation age resets to zero.

2.1.3 Rules for Rocks

- Rocks do not move and no animal can occupy their space. They act as permanent obstacles that constrain movement patterns.

2.1.4 Rules for Selecting Adjacent Cells

- Following clockwise order, number from 0 the possible P cells to where a fox or rabbit can move (adjacent north, east, south and west cells).
- Let G represent the current generation and (X, Y) represent the cell coordinates where a fox or rabbit is positioned, then the adjacent cell to be chosen is $(G + X + Y) \bmod P$. Assume the initial generation is number 0 and the world origin is $(0, 0)$. This formula creates a deterministic but pseudo-random selection pattern that varies based on position and time, ensuring reproducible yet diverse movement patterns.

2.1.5 Rules for Conflict Resolution

In each generation, rabbits move first followed by foxes. When several animals try to move to the same position, these rules apply:

- When two or more rabbits move to the same cell, only the one with the highest procreation age survives. All others die. This rewards older, more established individuals.
- When two or more foxes move to the same cell, only the one with the highest procreation age survives. If ages are equal, the least hungry fox is kept. This prioritizes survival of the fittest, where well-fed foxes have an advantage.

3 Implementation

3.1 Data Structures

The implementation uses three core data structures to represent the ecosystem state:

- **Cell:** Represents each grid cell with its `id` (Rabbit, Fox, Rock, or None), `age` (generations since birth or last procreation), `gens_without_food` (generations since last meal for foxes), and `gen_updated` (last generation this cell was modified). The `gen_updated` field enables conflict detection during parallel execution.
- **Environment:** Contains the complete simulation state including generation parameters (`gen_proc_rabbits`, `gen_proc_foxes`, `gen_food_foxes`), grid dimensions (`r`, `c`), current generation (`g`), and two cell matrices (`m` and `new_m`). The dual-matrix design prevents read-after-write hazards by separating the read source from the write destination during each generation.
- **ThreadState:** Defines the work partition for each thread with inclusive `start_x` and exclusive `end_x` boundaries. Thread states are pre-computed once during initialization and stored in the environment, eliminating redundant calculations across generations.

3.2 Parallel Strategy

Based on the parallel analysis, we implemented a domain decomposition approach using OpenMP. The grid is divided into horizontal bands assigned to different threads. We chose horizontal bands over vertical bands or block decomposition because they improve cache locality: processing consecutive columns in a row keeps recently accessed memory in cache, reducing cache misses and improving performance.

The band size is calculated to distribute work evenly:

```
int gap_space = 2*(n_threads - 1);
int t_rows = e.r - gap_space;
int b_size = t_rows / n_threads;
int remainder = t_rows % n_threads;
```

The remainder is distributed evenly across threads, with the first `remainder` threads receiving one additional row. Each thread's boundaries are calculated once during initialization.

3.3 Movement Logic

Animal movement is handled by `single_rabbit_move` and `single_fox_move` functions that implement the complete movement, procreation, and conflict resolution logic for a single animal.

3.3.1 Rabbit Movement

Rabbit movement follows this sequence:

1. Select direction using `select_rabbit_direction`, which filters empty adjacent cells and applies the selection formula.
2. Check if rabbit can procreate (`age >= gen_proc_rabbits`) and has a valid direction.
3. If moving to a cell already occupied by another rabbit in `new_m`, resolve conflict by comparing ages. If the current rabbit procreates, its effective age for comparison is 0, implementing the procreation reset edge case.
4. If winning the conflict, write to destination with incremented age. If procreating, spawn offspring at origin with age 0 and reset destination age to 0 (if no conflict).
5. If not moving, increment age in place.

3.3.2 Fox Movement

Fox movement is more complex due to eating and starvation:

1. Select direction using `select_fox_direction`, which prioritizes rabbit cells over empty cells.
2. Check starvation: if `gens_without_food >= gen_food_foxes - 1` and no rabbit is available, fox dies immediately without moving.
3. Determine if fox is eating by checking if destination in `m` (not `new_m`) contains a rabbit. This implements the multiple-foxes-eating edge case, ensuring all competing foxes recognize the eating opportunity regardless of processing order.
4. If moving to a cell already occupied by another fox in `new_m`, resolve conflict by comparing ages, then hunger if ages are equal. When procreating, use age 0 for comparison.
5. If winning, write to destination with incremented age and hunger set to 0 if eating, otherwise incremented. If procreating, spawn offspring at origin.

6. If not moving but another fox already moved to this cell (detected via `cell_equals`), resolve conflict using the same criteria. This handles cases where multiple foxes stay in place but one previously moved here.

3.4 Generation Update

The `next_gen` function orchestrates a complete generation update through the following sequence:

1. Copy the current state matrix `m` to `new_m` to initialize the next generation's state.
2. Process rabbits in parallel: each thread iterates through its assigned band (from `start_x` to `end_x`) and calls `single_rabbit_move` for each rabbit, writing results to `new_m`.
3. Process the 2-row gap regions sequentially between each pair of adjacent bands to handle potential cross-band conflicts for rabbits.
4. Copy `new_m` back to `m` to establish the post-rabbit state before foxes move.
5. Process foxes using the same parallel band processing followed by sequential gap processing.
6. Swap the matrix pointers (`m` and `new_m`) and increment the generation counter.

This sequence ensures: (1) all reads come from `m` while writes go to `new_m`, (2) rabbits complete before foxes begin, (3) band interiors are processed in parallel while gap regions are sequential, and (4) matrix pointers are swapped rather than copying data for the next generation.

3.5 Performance Optimizations

Beyond the core parallel strategy, we implemented several optimizations to improve performance:

3.5.1 Horizontal Band Decomposition

We chose horizontal bands over vertical bands or block decomposition to exploit cache locality. Since matrices are stored in row-major order in C, processing consecutive columns within a row keeps recently accessed memory in cache. This spatial locality reduces cache misses significantly compared to vertical bands, which would jump between non-contiguous memory locations, or block decomposition, which would have smaller contiguous regions.

3.5.2 Pre-computed Thread States

Thread boundaries are calculated once during environment initialization and stored in the `Environment` structure. This eliminates redundant calculations across all generations, as the same band assignments are used throughout the simulation. Without this optimization, we would recalculate band boundaries in every generation.

3.5.3 Pointer Swapping

At the end of each generation, we swap the `m` and `new_m` matrix pointers rather than copying the entire matrix data. This transforms an $O(r \times c)$ copy operation into an $O(1)$ pointer swap, eliminating unnecessary memory operations between generations.

3.5.4 Integrated Direction Calculation

Direction selection is computed on-demand within the movement functions rather than pre-computing all directions in a separate phase. While pre-computation would be parallelizable, it would require allocating and writing to a direction matrix, then reading from it during movement. The overhead of this extra allocation and memory traffic exceeded the benefit of parallelizing a relatively cheap computation that only reads from the current state.

3.5.5 Matrix Copy Parallelization

The `copy_cell_matrix` function uses `memcpy` for each row and can be parallelized with OpenMP when copying between generations. While the row-by-row `memcpy` is already efficient, parallelizing across rows provides additional speedup for large grids.

3.5.6 Persistent Thread Pool

The OpenMP threads are instantiated once at the beginning of the simulation, with the parallel region encompassing the entire generation loop rather than being created and destroyed for each generation. By placing the `#pragma omp parallel` directive outside the loop that iterates over `n_gen` generations, we eliminate the repeated overhead of thread spawning and joining. This transforms the thread creation cost from $O(n_{gen})$ to $O(1)$, as threads remain active throughout the simulation and only synchronize at barriers between phases rather than being recreated.

3.5.7 Parallelized Matrix Synchronization

The primary bottleneck in our initial parallel implementation was the synchronization phase, where boundary rows between blocks assigned to different threads were computed sequentially. We address this limitation by exploiting the independence of these boundary computations: since they operate on disjoint memory regions without data dependencies, they can be executed concurrently without requiring explicit synchronization primitives. This transformation eliminates the sequential bottleneck and enables full parallelization of the synchronization phase.

3.6 Validation

To ensure correctness of the parallel implementation, we employ a multi-faceted validation approach:

3.6.1 Sequential Consistency

The parallel implementation produces identical results to the sequential version. The `assert_environment_equals` function verifies that the final state matches the expected output by comparing grid dimensions, generation parameters, and the `id` of every cell in the matrix.

3.6.2 Deterministic Behavior

Running the same input multiple times with the same number of threads always produces identical results. This validates that our conflict resolution and gap region strategy correctly handle race conditions, ensuring no non-deterministic behavior from parallel execution.

3.6.3 Edge Case Testing

Test cases specifically target the edge cases identified in pre-implementation analysis: starvation without movement, procreation age resets during conflicts, and multiple foxes eating the same rabbit. These tests verify the implementation correctly handles subtle rule interactions.

3.6.4 Thread Scalability Validation

The implementation is tested with different thread counts to ensure correctness is maintained regardless of domain decomposition. The `thread_state_init` function includes validation that thread count doesn't exceed grid capacity, preventing invalid configurations.

4 Performance Evaluation

4.1 Experimental Setup

All experiments were conducted on a system equipped with an AMD Opteron 6380 processor (4 sockets, 8 cores per socket, 64 hardware threads total). The OpenMP implementation used was GCC's libgomp runtime, compiled with GCC using optimization flag `-O3`.

To ensure the measurements were reliable, each configuration was executed 10 times. The values shown represent a median value across all runs, excluding I/0 operations. All timing measurements were obtained using `omp_get_wtime()` for OpenMP parallel regions and standard C `clock()` function for sequential execution.

The benchmark evaluated five different problem sizes (5×5, 10×10, 20×20, 100×100, and 200×200) across five thread configurations (1, 2, 4, 8, and 16 threads).

4.2 Execution Time Analysis

Table 1 presents the execution times for the sequential and the OpenMP parallel implementations across all ecosystem grid sizes. Both solutions show polynomial growth in execution time according to the grid size. The results show that for smaller grid sizes (5×5, 10×10 and 20×20), parallelization introduces significant

Grid Size	T=1	T=2	T=4	T=8	T=16
5×5	**0.0000232**	0.0002396	—	—	—
10×10	**0.0004935**	0.0014302	0.0019243	—	—
20×20	**0.0213541**	0.0293156	0.0309025	—	—
100×100	3.6614286	3.5902368	2.9425550	**2.8442560**	3.3997429
200×200	14.2381925	12.4184460	11.0393203	**10.4215301**	11.2320350

Table 1: Execution times (seconds) for sequential and parallel ecosystem simulations

synchronization overhead, taking considerably more time to execute compared to the sequential solution. For the smaller grids, the overhead is so substantial that tests with higher thread counts were not performed. For larger grid sizes (100×100 and 200×200), parallelization improves execution time. However, the benefit is limited by the cost of synchronization. While performance gains are observed up to 8 threads, increasing to 16 threads degrades performance due to excessive synchronization overhead.

4.3 Speedup Analysis

The speedup was calculated using the formula: $S_p = T_1/T_p$. It measures the run-time improvement between parallel and sequential implementations, for each grid size and number of threads used. The results are demonstrated in Table 2. As previously stated, the parallel implementation only shows improvements for larger grid sizes (100×100 and 200×200), failing to match or improve performance for matrices of size 5×5, 10×10 and 20×20. This occurs because small grids fit entirely in cache, where sequential execution benefits from excellent spatial locality. Parallelization introduces thread creation overhead and cache coherence traffic that outweigh the minimal computational work available.

For large grids, the data exceeds cache capacity, causing frequent cache misses in sequential execution that significantly degrade performance. Here, the substantial computational work dominates the parallelization

Grid Size	T=1	T=2	T=4	T=8	T=16
5×5	1.00	0.0968	—	—	—
10×10	1.00	0.3451	0.2565	—	—
20×20	1.00	0.7284	0.6910	—	—
100×100	1.00	1.0198	1.2443	**1.2873**	1.0770
200×200	1.00	1.1465	1.2898	**1.3662**	1.2676

Table 2: Speedup comparison across grid sizes and thread counts

overhead, enabling speedup with up to 8 threads, where optimal workload distribution and improved per-thread cache utilization outweigh synchronization costs.

4.4 Parallel Efficiency Analysis

The efficiency was calculated using the formula: $E_p = S_p/p$. This indicates how well computational resources are utilized. The results in Table 3 reveal that smaller grid sizes (5×5, 10×10, and 20×20) exhibit poor

Grid Size	T=1	T=2	T=4	T=8	T=16
5×5	100.0	4.8	—	—	—
10×10	100.0	17.3	6.4	—	—
20×20	100.0	36.4	17.3	—	—
100×100	100.0	**51.0**	31.1	16.1	6.7
200×200	100.0	**57.3**	32.2	17.1	7.9

Table 3: Parallel efficiency (%) across configurations

thread efficiency due to the synchronization overhead and cache coherence traffic dominating the minimal computational work. For larger matrices (100×100 and 200×200), the parallel implementation achieves reasonable efficiency, with the 200×200 grid reaching 57.3% efficiency at 2 threads and maintaining 32.2% at 4 threads, demonstrating that substantial computational work effectively amortizes synchronization costs and benefits from reduced cache miss rates per thread.

4.5 Key Findings

4.5.1 Grid size determines parallelization viability

For grids smaller than 100×100, sequential execution outperforms all parallel configurations due to thread management overhead and sequential gap region processing dominating computation time. Small grids benefit from excellent cache locality in sequential execution that parallel implementations cannot match, with parallel versions taking 2-20× longer for 5×5 to 20×20 grids.

4.5.2 Optimal thread count is problem-dependent

The 200×200 grid achieves maximum speedup (1.3662×) at 8 threads with 17.1% efficiency, while the 100×100 grid peaks at 1.2873× speedup with 8 threads and 16.1% efficiency. Beyond 8 threads, performance degrades as the fraction of work in sequential gap regions increases relative to parallel band processing, reducing overall efficiency.

4.5.3 Cache behavior explains performance transitions

Small grids ($\leq 20 \times 20$) fit entirely in cache where sequential execution exploits spatial locality effectively. Large grids ($\geq 100 \times 100$) exceed cache capacity, causing frequent cache misses in sequential execution

that create opportunities for parallel speedup through better per-thread cache utilization despite sequential processing costs.

4.5.4 Sequential regions dominate overhead

Even at optimal configurations, efficiency remains below 58%, indicating that thread creation overhead, cache coherence traffic, and mandatory sequential gap region processing consume significant portions of potential parallel performance gains. The 2-row gap regions between bands, while necessary for correctness, represent unavoidable sequential work that increases proportionally with thread count (more threads = more gaps).

4.5.5 Movement ordering creates inherent sequential constraints

The requirement that all rabbits move before any fox begins moving requires a generation barrier between species processing phases. This represents an Amdahl's Law limitation where the sequential portion (species ordering + matrix copying) caps maximum theoretical speedup regardless of thread count.

4.5.6 Horizontal band decomposition maximizes cache efficiency

Choosing horizontal over vertical bands exploited row-major memory layout, improving cache hit rates by processing contiguous memory locations. This spatial locality optimization was critical to achieving positive speedup for large grids, as cache miss penalties would otherwise dominate execution time.

4.5.7 Sequential work fraction increases with thread count

In densely populated regions where multiple animals compete for the same destination cell, conflict resolution in gap regions becomes more expensive. More critically, as thread count increases, the number of 2-row gap regions grows (`n_threads` - 1 gaps), increasing the fraction of work that must be processed sequentially. This explains why speedup improvements plateau and eventually degrade: at 16 threads, 30 rows (15 gaps $\times$ 2 rows) must be processed sequentially for a 200×200 grid, representing 15% of the grid processed outside parallel regions.

4.5.8 Architecture-dependent thread management strategies

Testing on ARM-based Apple Silicon processors revealed unexpected performance characteristics that contrast with the AMD Opteron results. The persistent thread pool approach—where threads are created once and reused across all generations using barrier synchronization—performed optimally on x86 architecture. However, on Apple Silicon, an alternative implementation that creates and destroys threads for each generation phase achieved noticeably better performance. This suggests that ARM's thread scheduling and cache coherence mechanisms favor lightweight thread creation overhead over the barrier synchronization costs of persistent threads. The performance difference likely stems from Apple Silicon's efficiency cores and memory subsystem architecture, where frequent thread spawning incurs less penalty than maintaining idle threads across barrier points. This architectural sensitivity highlights that optimal parallel strategies are not universally portable and must be evaluated on target hardware.

4.5.9 Practical Recommendations

- $\mathbf{N} < 100 \times 100$: Use sequential implementation for better performance
- $100 \times 100 \leq \mathbf{N} < 200 \times 200$: Use 2-8 threads for modest speedup (1.02-1.29×)
- $\mathbf{N} \geq 200 \times 200$: Use 8 threads for optimal performance (up to 1.37× speedup)
- **Consider architecture**: On x86 systems, use persistent thread pools; on ARM-based systems like Apple Silicon, evaluate thread creation per generation

- **Monitor thread count impact**: Beyond 8 threads, sequential gap region overhead dominates, degrading performance
- **Consider workload characteristics**: Dense populations near band boundaries reduce parallel efficiency due to increased conflict resolution overhead in gap regions

4.6 Future Improvements

4.6.1 Dynamic Load Balancing Through Entity-Aware Partitioning

Our current implementation uses uniform horizontal band sizes, distributing rows evenly across threads regardless of computational load. This represents a significant bottleneck, as workload is determined by entity count rather than geometric area. Rows densely populated with animals require substantially more computation than sparse or empty rows, yet both receive equal thread time under uniform partitioning.

A more sophisticated approach would count entities per row during initialization and dynamically assign irregular block sizes to achieve balanced workload distribution. Threads processing entity-dense regions would receive fewer rows, while threads handling sparse regions would process more rows, equalizing actual computational work across threads. This entity-aware partitioning would reduce thread idle time at synchronization barriers and improve overall parallel efficiency.

4.6.2 Strategic Gap Placement for Reduced Sequential Overhead

The 2-row gap regions between bands constitute unavoidable sequential work that scales linearly with thread count. Our current uniform placement strategy positions gaps at regular intervals regardless of entity distribution, forcing sequential processing of potentially entity-dense boundary regions.

An improved strategy would analyze entity distribution and strategically position gaps in sparse regions of the grid where fewer animals exist. By preferentially placing band boundaries through empty or low-population areas, the sequential gap processing phase would handle fewer entities, reducing its computational cost. This optimization would be particularly effective for ecosystems with non-uniform spatial distributions, where animals cluster in certain regions while leaving others sparse.

Combined with entity-aware partitioning, this approach would address the load balancing bottleneck identified in our performance analysis, potentially improving efficiency beyond the current 57.3% maximum and enabling better scalability to higher thread counts.

References

[1] Conway, J. (1970). Conway's game of life. *Scientific American*.