Report
Ecosystem Simulation

Parallel Computing - 2025

Carlos Pombo Soares
up202509349
Rui Pedro Mendo Coelho
up202106772

1. Problem Overview

In this project we simulated an ecosystem with a initial sequential approach and then tried to parallelise the program with openMP. (MPI was not used because the goal of the project was to work with shared memory rather than distributed memory).

The simulated ecosystem works with the following constrains:

- Rabbits move first, followed by foxes.
- Rabbits try to move to adjacent empty cells. If no empty cells are available, they remain in place.
- Foxes try to move to adjacent cells containing rabbits. If no rabbits are nearby, they move to empty cells.
- Rocks do not move.
- Rabbits and foxes can spawn new creatures once they reach the reproduction age defined in the input.
- swap_worlds contains the new world with updated positions.
- Both foxes and rabbits move clockwise.

2. Sequential Implementation

We started with all the details for the sequential version as indicated by the professor. First, we defined the structure, how the matrices would be reset, and how rabbits could respawn. We decided to divide the project into multiple functions to maintain clarity.

`print_world()`	Displays the current state of the world grid for a given generation.
`init_world()`	Initialises all cells in the world grids as empty spaces.
`fill_world()`	Fills the world with initial objects (rabbits, foxes, rocks) from input.
`is_inside(int x, int y)`	Checks if the given coordinates are within the world boundaries
`move_rabbit(int x, int y)`	Handles rabbit behaviour: movement to adjacent empty cells, reproduction, and conflict resolution when two rabbits attempt to move to the same cell.
`move_fox(int x, int y)`	Processes fox behaviour: searching for rabbits to hunt, moving to empty cells if necessary, reproduction, hunger updates, and starvation handling.
`move_rabbits()`	Processes movement and reproduction of all rabbits in the current generation.
`move_foxes()`	Processes movement, hunting, and reproduction of all foxes in the current generation.
`copy_rabbits()`	Copies all rabbits from the current world to the new world.
`reset_new_world()`	Resets cells in the new world that don't contain rocks.
`swap_worlds()`	Swaps the current world with the new world for the next generation.
`output()`	Outputs the final state of the world with all remaining objects and their positions.

The sequential implementation was crucial for unlocking the parallel design.

3. Parallel Implementation

We started by identifying parts of the code with large loops and tasks that could be executed simultaneously, such as processing different rows (loop within a loop). Initially, we only used OpenMP pragmas, but after some tests, the results were not significantly improved as it created a single global lock, which was not advantageous.

We then decided to use omp_lock and associated one lock per cell, allowing fine-grained serialisation of writes, reducing conflicts and maximising performance. In the case that rabbits and foxes attempt to move to the same cell (mostly between rabbits, since rabbits move first), we use omp_lock at the start to prevent race conditions. Each thread processes 4 rows, and scheduling is dynamic, as some areas are denser in creatures than others.

In our parallel approach, cell writes are divided among threads; locks are used per cell to resolve race conditions; locks are released after each write to prevent deadlocks and manage memory properly; schedule(static) is used for uniform operations like initialising the world/matrix and schedule(dynamic) is used for dynamic operations like moving the animals.

3.1. Parallel Changes Introduced

3.1.1. Parallelisation of Independent Loops

In the sequential version, loops such as world initialisation, grid cleaning, or object counting were executed entirely linearly. Since these loops have no inter-iteration dependencies, it's safe and efficient to parallelise them. The parallelised stages were world initialisation (`init_world()`), initial copy of foxes, reseting new_world (`reset_new_world()`), final copy of rabbits between generations (`copy_rabbits()`) and object counting.

Sequential approach	Parallel approach

`for(x = 0; x < R; x++)`
`  for(y = 0; y < C; y++)`
`  world[x][y] = ...;`
	`#pragma omp parallel for schedule(static)`

3.1.2. Parallel Movement of Animals

The movement of rabbits and foxes (`move_foxes()`) is the most computationally intensive part of the simulation. Since animal distribution is not uniform, some threads may have to work more than others. For that reason, as to improve load balancing, we used dynamic scheduling.

Sequential approach	Parallel approach

`for(x=0; x<R; x++)`
`  for(y=0; y<C; y++)`
`  if(world[x][y].type == 'R')`
`  move_rabbit(x, y);`
	`#pragma omp parallel for schedule(dynamic, 4)`

2.3. Introduction of Locks to Ensure Consistency

In the sequential version, there is no risk of conflict because only a single execution flow exists. In the parallel version, multiple threads may attempt to write to the same cell in `new_world`. To overcome this issue and ensure that only one thread writes to a given cell at a time, a matrix of locks was created, with one lock per cell. However, the cost of using these locks is that we end up increasing thread synchronisation that results in lower speedup.

These solution was applied inside `move_rabbit()` for rabbit reproduction and movement to a new cell. It also was applied inside `move_fox()` for handling fox hunting, fox reproduction, hunger, age updates and movement to a new cell.

Parallel solution

`omp_set_lock(&cell_locks[new_x][new_y]);`
`new_world[new_x][new_y] = ...;`
`omp_unset_lock(&cell_locks[new_x][new_y]);`

2.4. Separation Between Current and Next World

In the sequential version, world and new_world already existed, but updates were simple and linear. In the parallel version this model becomes essential because each thread reads only from world (avoids read/write races) and all threads write into `new_world`, protected by locks.

3.2. Summary of improvements from sequential implementation:

- Most computationally heavy loops (movement, initialisation, copying, resetting) were parallelised. `#pragma omp parallel for` was used to parallelise loops and balance the workload (static for uniform work; dynamic for irregular work);
- Dynamic scheduling was introduced to handle unbalanced workload, particularly for grids with uneven animal distribution.
- The use of lock per grid cell ensured safe concurrent updates, avoiding race conditions when multiple animals move or reproduce into the same cell. `amp_lock_t / omp_set_lock / omp_unset_lock` protect concurrent writes to new_world and are essential for correctness;

4. Results

For both implementations we took the times of executing the simulations for the following inputs: input5x5, input10x10, input100x100, input100x100_unball01, input100x100_unball02, input200x200. For parallel implementation we tested for 1, 2, 4, 8 and 16 threads.

Input File	Sequential	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
input5x5	0.02640	0.13646	0.17570	0.20890	0.30278	0.58182
input10x10	1.39954	2.69817	3.33272	4.23079	5.87187	9.36204
input100x100	8264.25214	10075.34390	6745.88129	3932.32600	2533.53117	1913.93670
input100x100_unbal 01	4382.71439	8277.35250	5763.44025	3581.78796	2323.99271	1812.36194
input100x100_unbal 02	4380.88905	9720.80733	6253.03998	3805.88050	2485.13308	1933.16392
input200x200	33220.01623	39175.21043	24418.77394	13846.17596	8089.18049	5194.38260

Table 1 - Execution times (ms)

Input File	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
input5x5	0.1936	0.1502	0.1263	0.0872	0.0454
input10x10	0.5187	0.4200	0.3307	0.2383	0.1495
input100x100	0.8205	1.2249	2.1025	3.2639	4.3175
input100x100_unbal01	0.5295	0.7606	1.2235	1.8852	2.4174
input100x100_unbal02	0.4507	0.7008	1.1511	1.7621	2.2655
input200x200	0.8483	1.3608	2.3998	4.1093	6.3964

Table 2 - Speedups

5. Conclusions

As we expected, our parallelisation proved more effective in improving performance for medium or large grids, and not so much for small grids due to the parallel overhead (e.g., input5x5: 0.1936 → 0.0454; performance worsened with thread number increase).

We can say that maintaining separate read-only and write-only worlds further contributed to safe and reliable parallel updates without race conditions and also, that dynamic scheduling was essential in helping to balance the workload, since animal distribution is likely to vary across the grid and enabled us to reduce the idle time among the threads.

However, even with these upgrades, there was still some limitations. When compared to more balanced grids of the same size, unbalanced worlds led to lower efficiency, because of the irregular workloads and increased locking. For example, for 16 threads, input100x100_unball01 (unbalanced) showed a lower speedup (2.4174) when compared to input100x100 (4.3175). Moreover, and although the introduction of per-cell locks ensured correct behaviour, it also added synchronisation costs that limited the maximum achievable speedup.