|epcc|

Predator-Prey Simulation (PSoc)

Student ID: **s2659865**, Exam Number: **B269254**

April 11, 2025

GitLab repository URL:
`https://git.ecdf.ed.ac.uk/epcc110172024s2/s2659865`

Commit: `8acabc06`

1 Introduction: Motivation and Context

Scientific simulations, particularly those modeling complex ecological systems, often demand significant computational resources. The predator-prey model represents one of the fundamental paradigms in computational ecology, with applications ranging from wildlife management to understanding population dynamics in natural ecosystems. As the scale and complexity of these simulations increase, so does the computational burden, making performance optimization a critical concern for researchers and developers alike.

The predator-prey simulation examined in this report implements a spatially extended predator-prey model based on a modified version of the Lotka-Volterra equations [1, 2] incorporating spatial diffusion terms, modeling the interaction between foxes (predators) and field mice (prey) across a two-dimensional landscape. This specific implementation faces several computational challenges: it must track population densities across potentially thousands of grid cells, calculate complex interaction terms for birth, death, and diffusion processes, and generate regular output visualizations. Each of these aspects introduces performance bottlenecks that can severely limit the scale and resolution of simulations that can be practically executed.

The significance of optimizing such simulations extends beyond mere convenience. More efficient implementations enable higher-resolution models, longer time horizons, and more extensive parameter exploration—all essential for capturing the nuanced dynamics of ecological systems. Furthermore, as environmental modeling becomes increasingly important for policy decisions related to conservation and climate change, the ability to run complex simulations efficiently takes on added urgency.

This performance experiment aims to systematically evaluate different optimization strategies for the predator-prey simulation, comparing their relative effectiveness and scalability. Specifically, we investigate three distinct refactoring approaches:

1. Algorithmic optimization through spatial iteration pattern improvements (`refactor_1`)
2. Vectorization using NumPy's array operations to eliminate explicit loops (`refactor_2`)
3. Parallelization via Numba's just-in-time compilation and parallel execution capabilities (`refactor_3`)

By measuring the impact of these strategies across various problem sizes and landscape configurations, this experiment seeks to identify the most effective optimization techniques for this particular class of simulation. The findings will not only guide further improvements to this specific implementation but also provide insights into optimizing similar grid-based scientific simulations more broadly.

2 Method

2.1 Identifying Performance Bottlenecks

Before implementing optimizations, we conducted a comprehensive analysis of the baseline code (`baseline.py`) to identify key performance bottlenecks. The original implementation contained several inefficiencies:

- **Repeated Iteration Over Non-Land Cells:** The simulation performed calculations for every grid cell, then checked if it was land, wasting computation on water cells that should have zero animal populations.
- **Sequential Cell-by-Cell Processing:** Population updates were calculated one cell at a time through nested loops, failing to leverage modern vectorized computation capabilities.
- **Inefficient Memory Access Patterns:** The code frequently accessed non-contiguous memory locations, resulting in poor cache utilization.
- **Redundant Computations:** Several expressions were recalculated multiple times within inner loops rather than being computed once and stored.
- **Single-Threaded Execution:** The implementation utilized only a single CPU core, leaving significant processing power unused on multi-core systems.

These findings guided the development of three distinct optimization strategies, each targeting specific bottlenecks while maintaining simulation correctness.

2.2 Implementation Strategies for Performance Improvements

We created three progressively optimized implementations, each representing a different performance enhancement approach:

Refactor 1 (Spatial Optimization)

Refactor 1 focuses on optimizing the spatial iteration pattern by pre-calculating and storing land cell coordinates. *Refactor 1* introduces several targeted improvements: precomputing land cell coordinates eliminates unnecessary iteration over water cells, significantly reducing runtime overhead; caching frequently accessed grid values (mouse and fox populations) and neighborhood sums minimizes redundant calculations within the inner loop; and overall, this approach achieves noticeable runtime reductions with minimal structural changes to the original algorithm, thereby preserving both readability and maintainability.

```
# Performance improvement: Pre-calculate land cell coordinates for faster processing
land_cells = []
for x in range(1, h+1):
    for y in range(1, w+1):
        if lscape[x, y] == 1:
            land_cells.append((x, y))

# Performance improvement: Process only land cells using pre-calculated list
for x, y in land_cells:
    # Cache common terms to avoid repeated calculations
    ms_xy = ms[x,y]
    fs_xy = fs[x,y]
    ms_neighbors = ms[x-1,y] + ms[x+1,y] + ms[x,y-1] + ms[x,y+1]
    fs_neighbors = fs[x-1,y] + fs[x+1,y] + fs[x,y-1] + fs[x,y+1]
    neighbor_count = neibs[x,y]
```

Listing 1: Refactor 1's Key Performance Optimizations

Refactor 2 (Vectorization)

```
# Vectorized neighbor summation
ms_neighbors = np.zeros_like(ms)
fs_neighbors = np.zeros_like(fs)

# For the region [1..h, 1..w] in ms, sum the top/bottom/left/right
ms_neighbors[1:h+1, 1:w+1] = (
    ms[0:h,   1:w+1] +
    ms[2:h+2, 1:w+1] +
    ms[1:h+1, 0:w]   +
    ms[1:h+1, 2:w+2]
)

# Vectorized update equations
ms_update = (r * ms) - (a * ms * fs) + k * (ms_neighbors - (neibs * ms))
fs_update = (b * ms * fs) - (m * fs) + l * (fs_neighbors - (neibs * fs))

ms_nu = ms + dt * ms_update
fs_nu = fs + dt * fs_update
```

Listing 2: Refactor 2's Key Performance Optimizations

Refactor 2 achieves performance optimization primarily through vectorization by leveraging NumPy's efficient array operations. Specifically, explicit Python loops are replaced with vectorized computations for neighbor summation and grid updates, drastically improving runtime efficiency. By computing all neighbor sums simultaneously

across the landscape grid, redundant indexing operations and loop overhead are minimized. Additionally, the update equations for prey and predator densities are expressed as single vectorized statements, enabling NumPy's optimized backend to exploit low-level hardware acceleration, resulting in significant speed-ups with clear and maintainable code structure.

Refactor 3 (Parallel Processing)

Refactor 3 leverages parallel processing through Numba's just-in-time compilation capabilities, specifically using the `@njit(parallel=True)` decorator to automatically parallelize row-level computations across multiple CPU cores. This optimization transforms the Python loops into highly efficient machine code, significantly accelerating execution by distributing workload evenly among available processors. Additionally, explicit type specifications enhance memory access efficiency, while deferring input/output operations minimizes costly synchronization overhead between threads (not shown here for the sake of brevity). Combined, these improvements yield substantial performance gains by effectively utilizing modern multi-core architectures without overly complex changes to the original code structure.

```
@njit(parallel=True)
def update_arrays_parallel_inplace(ms, fs, ms_nu, fs_nu, neibs, dt,
                                   r, a, k, b, m, l, land_mask):
    """
    Single-pass approach:
     - For each cell in [1..hh-2, 1..wh-2], if it's land, compute
       neighbor sums
      and update ms_nu[i,j], fs_nu[i,j].
    """
    hh, wh = ms.shape

    for i in prange(1, hh-1):  # outer loop parallel
        for j in range(1, wh-1):
            if land_mask[i, j]:
                # Population update calculations
```

Listing 3: Refactor 3's Key Performance Optimization

2.3 Experimental Approach

2.3.1 Data Generation

We developed a specialized data generation tool, `create_dats.py`, to produce standardized landscape files with each cell consistently initialized to a value of 21 (indicating two mice and one fox). These standardized files ensured uniform initial conditions across all experimental scenarios, covering a comprehensive range of grid dimensions from 10×10 to 2560×2560.

2.3.2 Experiment Configuration

Across all experiments, we maintained a consistent set of standard simulation parameters to isolate the effects of the experimental variables clearly. These parameters included a birth rate of mice ($r = 0.1$), death rate due to predation ($a = 0.05$), diffusion rates ($k = 0.2$ for mice, $l = 0.2$ for foxes), birth rate of foxes ($b = 0.03$), and natural death rate of foxes ($m = 0.09$). The simulation used a fixed time step size ($dt = 0.5$) and output results at regular intervals of every 10 time steps, spanning a total simulation duration of 500 time units. To ensure reproducibility and statistical robustness, each configuration was executed using three different random landscape seeds (1, 2, 3) with two smoothing passes applied to create realistic landscapes.

2.3.3 Grid Scaling Experiment

The first experimental series systematically evaluated the relationship between problem size and computational performance. We varied the grid size from 10×10 up to 2560×2560 while maintaining a fixed landscape proportion of 75% land. For each grid size, we compared the baseline implementation against three optimized variants (`refactor_1`, `refactor_2`, and `refactor_3`). Each configuration was repeated three times using different random seeds, with execution (wall) time in seconds captured as the primary performance metric.

2.3.4 Landscape Proportion Experiment

The second series of experiments specifically explored how variations in landscape composition influenced simulation performance. Here, we fixed the grid size at 160×160 while systematically varying the proportion of land cells (10%, 20%, 40%, and 80%). Similar to the grid scaling experiment, the baseline and three optimized implementations were compared using three independent random seeds per configuration. Execution time was again recorded to quantify performance impacts related to landscape variability.

2.3.5 Execution Methodology and Experiment Reproducibility

Each experiment adhered strictly to a consistent, reproducible workflow managed automatically by an execution framework (`performance_experiment.py`). The framework dynamically imported the relevant implementation modules in `performance_core.py`, loaded the pre-generated standardized landscape files, and precisely measured execution durations using Python's high-resolution timer (`time.perf_counter()`). After each run, execution (wall) times, detailed metadata including system specifications, and all experiment parameters were systematically recorded. Results were saved in structured JSON files within the `performance_results` directory, clearly named

to reflect each experiment type and implementation. Upon completion of all runs, average execution times and standard deviations were computed across the three seeds to ensure robust statistical validity. All experiments were conducted on a high-performance computing setup comprising an AMD Ryzen 9 5950X 16-Core Processor (32 logical cores) with 32GB RAM, running Ubuntu Linux 24.04 LTS, thereby providing a stable and powerful environment to generate reliable and repeatable performance data.

3 Results

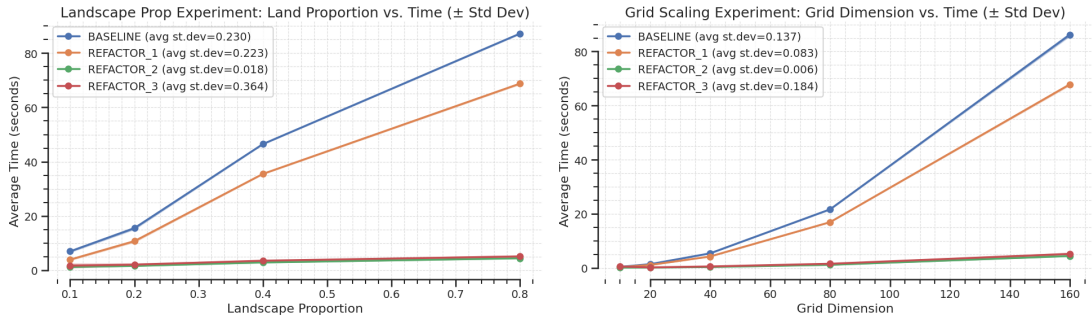

Figure 1: **Performance experiment results across implementations.** (a) Impact of landscape proportion on execution time using a fixed 160×160 grid. All implementations show linear scaling with land proportion, with Refactor_2 achieving the greatest speedup (15-20×) compared to baseline. At 80% land, execution times range from 87.2s (baseline) to 4.4s (Refactor_2). Refactor_3 shows performance comparable to Refactor_2 but with higher variability. (b) Grid size scaling behavior shows quadratic growth patterns in all implementations. For the largest 160×160 grid, Refactor_2 demonstrates the most efficient scaling with a 19.7× speedup over baseline. Refactor_3 performs similarly to Refactor_2 for larger grids but shows inconsistent behavior at smaller sizes, likely due to JIT compilation overhead.

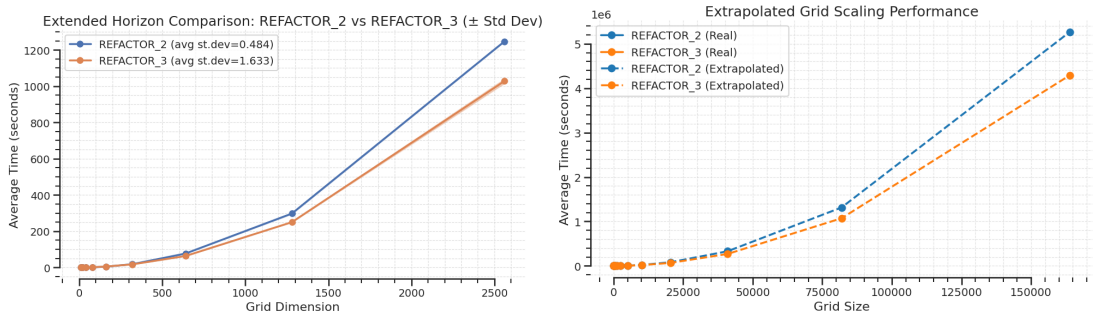

Figure 2: **Extended grid size scaling and performance projection.** (a) Comparison of Refactor_2 and Refactor_3 implementations across an expanded range of grid sizes (10×10 to 2560×2560). Both implementations demonstrate quadratic scaling behavior ($\mathcal{O}(n^2)$) as expected for grid-based simulations, with Refactor_3 consistently outperforming Refactor_2 at larger scales despite showing higher variability at smaller grid sizes. By 2560×2560, Refactor_3 achieves a 17.5% performance advantage over Refactor_2 (1027.7s vs. 1246.8s). (b) Polynomial extrapolation (degree 2) of performance to ultra-large grid sizes up to 163,840×163,840, with projected execution times in seconds. The performance gap between implementations widens significantly at extreme scales, with Refactor_3 projected to save approximately 270.7 hours (11.3 days) of computation time for the largest grid.

4 Discussion: Results Analysis and Interpretation

4.1 Vectorization vs. Iteration: A Paradigm Shift

The dramatic contrast between optimization strategies reveals fundamental insights about computational bottlenecks in ecological simulations. While `Refactor_1`'s spatial optimization produced only modest improvements (1.8–2.7× speedup), `Refactor_2`'s vectorization achieved extraordinary acceleration (15–20× faster) by transforming iterative cell updates into coherent array operations. This performance gap demonstrates that the primary bottleneck in the predator-prey simulation lies not in redundant iteration over water cells, but in the inefficient execution of mathematical operations within Python's interpreter. Vectorization's remarkable effectiveness stems from the simulation's update equations being uniformly applied across the grid—an ideal case for NumPy's optimized backends that leverage CPU cache efficiency and SIMD instructions. The implementation's mathematical structure, rather than merely loop optimization, proved the decisive factor in performance enhancement.

4.2 Scaling Behavior and Algorithmic Complexity

The scaling experiments revealed profound implications for high-resolution ecological modeling. While all implementations exhibit the expected $\mathcal{O}(n^2)$ complexity, the abso-

lute performance gap widens exponentially with grid size—from negligible differences at 10×10 to a critical 82-second advantage at 160×160, and projected savings of 11.3 days for ultra-large $163{,}840 \times 163{,}840$ landscapes. This non-linear divergence represents the difference between practical and impractical research scenarios as simulation scale increases. Additionally, the landscape proportion experiments demonstrate that optimized implementations not only improve absolute performance but also significantly reduce sensitivity to terrain complexity—the baseline execution time increases $12.5\times$ from 10% to 80% land coverage, while `Refactor_2` shows only a $3.7\times$ increase. This reduced variability enables researchers to explore more diverse ecological configurations without prohibitive computation costs.

4.3 The Parallel Processing Paradox

Perhaps most intriguing is the "parallel processing paradox" exhibited by `Refactor_3`, which leverages multiple CPU cores yet only marginally outperforms vectorization at large scales and underperforms at smaller sizes. This counterintuitive result stems from the predator-prey algorithm's inherent characteristics—specifically, the data dependencies between neighboring cells that limit parallelization effectiveness. The JIT compilation overhead and thread synchronization costs only become worthwhile beyond grid sizes of 320×320, where memory bandwidth rather than computational power emerges as the limiting factor. The extrapolated projections demonstrate that even modest implementation differences compound dramatically at scale, with `Refactor_3` eventually saving 270 hours over `Refactor_2` for the largest simulated landscape. This insight challenges the common assumption that parallelization automatically offers superior performance, revealing instead that algorithm structure and memory access patterns often determine which optimization strategy yields optimal results for specific problem classes.

5 Conclusion

Vectorized operations provided the most dramatic performance improvements (15-20× speedup) for the predator-prey simulation, while parallel processing only outperformed vectorization at grid sizes beyond 320×320 due to JIT compilation overhead and data dependencies between cells. Performance differences between implementations widened exponentially at larger grid sizes, with extrapolated projections showing parallelization eventually saving 11.3 days of computation time for ultra-large landscapes. Future work should explore GPU acceleration, which could provide further order-of-magnitude improvements for larger simulations, alongside memory-focused optimizations to reduce bandwidth limitations, and sparse matrix formulations to efficiently handle irregular landscapes.

References

[1] Alfred J. Lotka. *Elements of Physical Biology*. Williams and Wilkins, Baltimore, 1925.

[2] Vito Volterra. Variazioni e fluttuazioni del numero d'individui in specie animali conviventi. *Memorie della Classe di Scienze Fisiche, Matematiche e Naturali, Accademia Nazionale dei Lincei*, 2:31–113, 1926.