COP4521 Programming Assignment 3: Parallel 2D Prisoner's Dilemma Simulation

Objectives:

- Practice parallel programming using Python multiprocessing and Domain Decomposition
- Practice parallel programming using OpenMP (extra points)

Description:

In this assignment, you will develop parallel programs to simulate the Prisoner's Dilemma game (as described in Programming Assignment 1) on a 2-dimensional grid. In the simulation, each cell in the 2D grid represents an agent that repeatedly interacts with its neighbors over many time steps. The simulation is designed to study the evolution of cooperation and strategy adaptation in a spatially distributed population.

Each agent considers the agents in the cells to the north, south, east, and west of its cell as neighbors and plays the game with them at each time step. An agent located in the middle of the grid interacts with all four neighbors (north, south, east, and west), while an agent on the boundary of the grid may interact with fewer than four neighbors.

At each time step, every agent:

- Maintains its current strategy (cooperate or defect),
- Uses this strategy in games with its neighbors,
- Sums the rewards from all games played during the time step, and
- Adopts the strategy of the neighbor (or itself) with the highest reward.

In the case of a tie for the highest reward, the agent adopts the strategy in the following priority order: itself, then the north neighbor, followed by the south, east, and west neighbors.

You will be provided with a sequential implementation of the simulation in Python. Your task is to develop a parallel implementation using Python's multiprocessing module with domain decomposition.

Implementation details:

You can run `seq_assignment3` on **linprog** using the following command:

```
<linprog6:515> python3 seq_assignment3.py grid_size iterations
```

For example, the following command runs the simulation on a 10x10 grid for 10 iterations:

```
<linprog6:515> python3 seq_assignment3.py 10 10
```

There are five different grid initialization routines in `seq_assignment3.py` for testing purposes. Your parallel version of the simulation should support the following command-line format:

```
<linprog6:515> python3 your_assignment3.py grid_size iterations num_of_processes
```

For example, the command below runs the parallel simulation on a 10x10 grid for 10 iterations using 4 processes:

```
<linprog6:515> python3 your_assignment3.py 10 10 4
```

The screen output as well as the output file of each simulation run by your parallel version **must be identical** to that of the sequential version.

Due time: October 13, 2025, 11:59pm.

Submission: Name your program using the format: **lastname_firstinitial_assignment3.py.** If you complete the extra credit using OpenMP, name that program: **lastname_firstinitial_assignment3_omp.c.** Place both files (if applicable) into a **.tar** archive and submit the tar file on Canvas**.**

Grading (60 points total):

- A program receives at most 6 points if it generates a runtime error.
- Include the basic header (template available on the course website) for the assignment. Name your program lastname_firstinitial_assignment3.py. Report the timing and speedup (compared to the sequential version) of your program using 1, 2, 4, 8, and 16 processes. (10 points)
- Your parallel implementation must produce the same output for a 10x10 grid and 20 iterations using 2 processes. (10 points)
- Your parallel implementation must produce the same output for a 10x10 grid and 20 iterations using 4 processes. (10 points)
- Your parallel implementation must produce the same output for a 1024x1024 grid and 20 iterations using 2 processes, and the program must achieve a 20% speedup compared to the sequential program. (10 points)
- Your parallel implementation must produce the same output for a 1024x1024 grid and 20 iterations using 4 processes, and the program must achieve a 20% speedup compared to the sequential program. (10 points)
- Your parallel implementation must produce the same output for a 1024x1024 grid and 20 iterations using any number of processes (nprocs ≤ grid_size or ≤ 16), and the program must achieve a 4× speedup compared to the sequential program. (10 points)
- Your program will be tested on another grid size. **20 points will be deducted** if the code passes the 10x10 and 1024x1024 grid tests but fails this test.

- **8 extra points** for a program that earns all 60 regular points, with each process using only O(grid_size × grid_size / nprocs) memory.
- **8 extra points** for an OpenMP implementation that works correctly for all grid sizes and achieves at least a 4× speedup for some grid size. To receive the extra points, you must also specify how the speedup was achieved (compared to either a sequential C version or your OpenMP code using 1 thread).
- **Extra points:** The first person to report a bug in the sample program will receive **3 extra points**.

Note:

- This program can be extremely challenging for those who have never written a distributed-memory parallel program.
- The structure of the parallel code is very similar to the Jacobi code that we discussed in detail in class.
- Both actionGrid and rewardGrid need to be distributed in each iteration. The order should be (1) communicating actionGrid, (2) computing rewardGrid, (3) communicate rewardGrid, (4) compute strategy adoption. count1 and count2 need to be summed across processes. Look at the Jacobi code to see how this can be done. Output to the file needs to be done by one process. See the Jacobi code for an example.
- The extra point for reducing memory usage in the processes is quite challenging.
- Once you have a C sequential implementation of the simulation, the OpenMP version is straightforward. All should try to earn the extra point for the OpenMP implementation.