CS3210 Assignment 1 Report

Students:

- Pham Ba Thang (E0550373)
- Nguyen Quy Duc (E0851456)

Program Implementation

Program Design

This Invasion of Mugland simulation is done using OpenMP and C++. The basic idea is going through every `generation` (starting from the given world), and at each generation, we loop through every `cell` and try to calculate the state of that cell in the next generation as well as increase number of `deathToll` if there is a death due to fighting or invasion on that cell. We can parallelize the generation of next state of each cell by using the OpenMP directive `#pragma omp parallel for reduction(+ : deathToll) collapse(2)`.

Parallel Strategy

To parallelize the program, we use the OpenMP directive `#pragma omp parallel for reduction(+ : deathToll) collapse(2)` to parallelize the loop and sum the number of death tolls.

- The number of threads is set by using `omp_set_num_threads(nThreads)`, where `nThreads` is the number of threads passed in as a command line argument.
- `#pragma omp parallel for` is to parallelize the loop where each thread will be assigned a chunk of the loop to work on.
- `reduction(+ : deathToll)` is to sum the number of death tolls from each thread into the `deathToll` variable.
- `collapse(2)` to collapse the nested loop into one loop so that we can better distribute the work to each thread, which leads to better performance.

Implementation details and special considerations

Minimize false sharing: Since `world` is a 2D vector that is shared between threads, we need to minimize writing to `world` as much as possible to avoid false sharing. Instead of initializing `world` with 0s and updating the value of each cell, we can initialize `world` with the value of the previous generation and only update the value of the cell if it changes.

Use of omp reduction to sum the dead tolls: Using critical section to update `deathToll` is not a good idea as it will defeat the purpose of parallelization as only one thread can access the critical section at a time. Summing the dead tolls in each thread, storing them in an array and summing the array at the end of the loop is also not a good idea as it will lead to false sharing. Therefore, we use `reduction(+ : deathToll)` to sum the dead tolls in each thread into `deathToll` variable.

Description, visualization, and data on your execution

How to reproduce the results

- We write a test script to generate 3 test cases (small, medium, and large)
    - small: 7x7 matrix with 10 generations
    - medium: 20x20 matrix with 100000 generations
    - large: 50x40 matrix with 100000 generations
- We test the program with 1, 4, and 8 threads on 2 different machines i7-7700 (8 logical cores) and Xeon Silver 4114 (20 logical cores)
- Command used:

```
./run_slurm.sh ./large_input.in ./large_input.out <num_threads>
```

The performance

The performance is measured by running the program 3 times for each test case using `perf stat`. The result is shown in the table below:

i7-7700 (8 cores)

Number of threads	7x7 matrix/10 gens (s)	20x20 matrix/100000 gens (s)	50x40 matrix/100000 gens (s)
1 thread	0.01036 +- 0.00210	3.9298 +- 0.0281	19.6424 +- 0.0391
4 threads	0.005949 +- 0.000223	1.29737 +- 0.00547	6.362 +- 0.1060
8 threads	0.006346 +- 0.000443	1.45810 +- 0.00721	6.6376 +- 0.0518

Xeon-Silver (20 cores)

Number of threads	7x7 matrix/10 gens (s)	20x20 matrix/100000 gens (s)	50x40 matrix/100000 gens (s)
1 thread	0.01807 +- 0.00620	3.91016 +- 0.00738	19.6627 +- 0.0335
4 threads	0.01538 +- 0.00326	1.3341 +- 0.0400	5.9537 +- 0.0267
8 threads	0.01648 +- 0.00307	1.5218 +- 0.0430	1.3341 +- 0.0400

Observation:

- In general, if we have a big world, the more threads we use, the faster the program runs since we can distribute the work to more threads.
- However, if we take a look at the medium size test cases for both machines, it's clearly that 4 threads perform better than 8 threads. The more threads doesn't mean better performance (faster execution time). This is due to the overhead of creating threads, context switching and synchronization overhead between the threads.

Matrix size vs number of generations

- **The aspect that affects the speed up of the program the most is number of generations**. Because in the implementation, we can only parallelize the work done in each iteration (generation), but no parallelism between different generations.
- So as the number of generations become very large (>1000000) for decent large matrix (50x50), the runtime is still slow.

Performace Optimization

One performance optimization is to **reduce the number of calculation for each cell**. Instead of checking for every condition (fighting, reproduction, overpopulation, underpopulation, survive, and invasion), we can speed up by changing the order of checks and remove redundant checks. For example:

- Whenever invasion happens, we can check the invasion plan first as the next generation cell value will be guaranteed to be the value of the invader
- Instead of checking different conditions separately, we performs all the check in one go given the counts of neighbor values
- We actually don't need to check for survival as this will always be true if all other checks (fighting, reproduction, and overpopulation) fail

Another performance optimization is to use further minimize false sharing by using **loop tiling**. Instead of looping through the whole matrix, we can loop through the matrix in tiles of size `tileSize` and update the value of the cells in the tile. This reduces the likelihood of multiple threads writing to the same cache line within the world vector.