Assignment 1
Parallel Implementation of Game of Invasions

CS3210 – 2021/22 Semester 1

3 Sep 2021

Learning Outcomes

This assignment aims to provide an introduction to parallel programming with shared memory paradigms (pthreads and OpenMP). We hope it reinforces your understanding of the process of parallelizing a sequential algorithm and gives a fun and satisfying introduction to parallel programming.

1 Problem Scenario

In this assignment, you will parallelize a modified version of Conway's Game of Life called Game of Invasions. The first section introduces the problem, the program specifications and the starter code. The second section addresses administrative issues for your submission.

Notagain is a flat two-dimensional world where life evolves from generation to generation according to a set of rules. Every so often, Notagain gets invaded by hostile aliens who then take up residence there (if their invasion succeeds), possibly with an uneasy truce or occasional fighting among the different factions. After a while, the surviving residents get invaded by other hostile aliens and the game of invasions continues.

1.1 Simulating the Game of Invasions

A simulation consists of zero or more generations (world states). A generation can be represented with an N x M rectangular grid where each grid cell in the world is either dead or contains exactly one lifeform. Each lifeform belongs to one of nine different factions (numbered 1–9), and every faction is hostile to every other faction. A friendly neighbor is one from the same faction. Likewise, a hostile neighbor is one from a different faction.

To evolve from one generation, G, to the next generation, $G + 1$, every cell in G will interact with its eight neighbors (the cells horizontally, vertically and diagonally adjacent to it) to determine its next state in $G+1$, according to the rules shown in Table 1. Note that cells along the world border may have no neighbors. That is, this simulation has no world wraparound.

1.1.1 Rules

Table 1 shows the rules of the game. The rules are applied at generation $G + 1$ for each cell considering the state of the cell and neighbours at generation G. At most one rule can be applied, as the rules are

Rule	Description
Fighting	If a live cell has at least 1 hostile neighbor, then it will die from fighting.
Underpopulation	If a live cell has fewer than 2 friendly neighbors (i.e. from the same faction) and no hostile neighbors, then it will die from underpopulation.
Overpopulation	If a live cell has at least 4 friendly neighbors and no hostile neighbors, then it will die from overpopulation (too much friendship is also unhealthy).
Survival	If a live cell has 2 or 3 friendly neighbors and no hostile neighbors, it will live on to the next generation.
Reproduction	If a dead cell has exactly 3 neighbors of the same faction, it will become alive in the next generation. If multiple factions contend to reproduce in the same cell, the higher numbered faction wins.

Table 1: Rules of Game of Invasions

mutually exclusive. If you use the sample implementation provided, you should not have issues related to the correctness of the game evolution.

1.1.2 Invasions

At pre-specified world states, an invasion may occur (at most one per world state; Notagain is not that unfortunate). If an invasion happens, 0 or more invaders will land on certain cells in the world, killing any lifeform that they land on, including those of their own faction.

When determining generation $G + 1$, a cell in generation G that is to be invaded in generation $G + 1$ will interact as per normal with its neighbors and then the invader will land. In other words, an invader landing is the final thing that happens when determining a new world state.

1.1.3 Death Toll

The program for this assignment should count the death toll due to fighting for a given simulation of Notagain. The count should include all cells that:

- died due to the Fighting rule specified in the Rules section 1.1.1.
- died because an invader landed on them — even those of the same faction

1.1.4 Examples

This section provides examples showing the simulation rules in action. Table 2 shows simple examples of each of the basic rules in action. A dead cell has no number while a live cell contains the number of the faction that that lifeform belongs to. The focus is on what happens to the highlighted cell(s) from Generation G to Generation $G + 1$. Assume that the cells shown are part of a larger world. (It may seem that multiple rules can apply in the provided examples, but this is because we show only the cells that are essential to understand the application of each rule.)

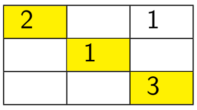
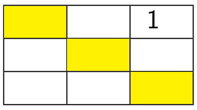
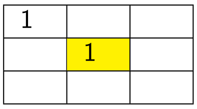
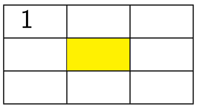
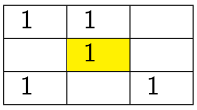
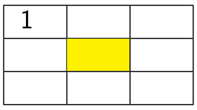
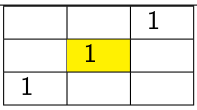
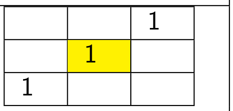
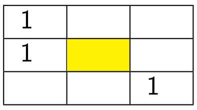
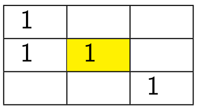
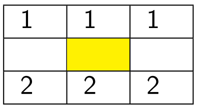
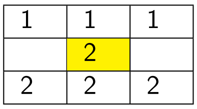
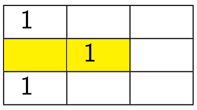
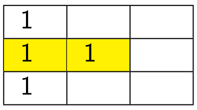
Rule	Generation G	Generation $G + 1$
Fighting		
Underpopulation		
Over Population		
Survival		
Reproduction		
Reproduction Tie-breaker		
Survival + Reproduction		

Table 2: Examples of Applying the Rules

The example in Table 4 shows a simulation of a 7×7 world for five new generations after the starting generation, with an invasion taking place as specified below. At generation 2, the invaders shown in Table 3 will land.

				3	3	3
	3	3				
	3	3				

Table 3: Invasion at generation 2 in the Game of Invasions Example

Generation 0

	1					
		1				
1	1	1				
					2	2
					2	2

Generation 1

1		1				
	1	1				
	1				2	2
					2	2

Generation 2

		1		3	3	3
1		1				
	3	3			2	2
	3	3			2	2

Generation 3

					3	
	1				3	
	1				3	
					2	2
	3	3			2	2

Generation 4

				3	3	3
					2	2

Generation 5

					3	
					3	
					3	

Table 4: Game of Invasions Example

Highlighted cells in the simulation show the live cells about to be killed due to fighting. These count toward our death toll due to fighting, which adds up to 8.

1.2 Inputs and Outputs

Your program should accept three command-line arguments.

- **The first** is a path to an input file with the simulation parameters. The program should then run the simulation, and compute the death toll due to fighting.
- Output a single integer representing the death toll due to fighting to the path specified by **the second** command-line argument.
- **The third** command-line argument represents the number of threads used in your OpenMP/threads implementations. Varying this argument should allow running these implementations with different numbers of threads. In the provided sequential implementation, this command-line argument is ignored.

The input file strictly follows the structure shown below:

- `N_GENERATIONS`: number of new generations to be simulated. This does not count generation 0: the starting world. If `N_ITERATIONS = 10`, then 10 new world states will be generated. $0 \leq \texttt{N\_GENERATIONS} \leq 1,000,000$
- `N_ROWS`: number of rows in the 2D world. $1 \leq \texttt{N\_ROWS} \leq 1,000,000$
- `M_COLS`: number of columns in the 2D world. $1 \leq \texttt{M\_COLS} \leq 1,000,000$
- `STARTING WORLD`: the layout of generation 0 (must be `N_ROWS` × `M_COLS`). Columns are separated by spaces; rows are separated by newlines.
- `N_INVASIONS`: number of invasions in the simulation. $0 \leq \texttt{N\_INVASIONS} \leq 1,000$
- `N_INVASIONS` repetitions of the following (sorted in ascending order by `INVASION_TIME`):
  - $\texttt{INVASION\_TIME}_i$: the generation number that the i^{th} invasion will land at. $1 \leq \texttt{INVASION\_TIME}_i \leq \texttt{N\_GENERATIONS}$, where each value $\texttt{INVASION\_TIME}_i$ is distinct
  - $\texttt{INVASION\_PLAN}_i$: basically a world layout (`N_ROWS` × `M_COLS`) with a 0 at each cell where nothing is landing and a value from 1 to 9 inclusive (corresponding to the faction of the lifeform) where something is landing

Your output file should strictly follow the structure shown below:

- `N_DEATH_TOLL`: number of deaths due to fighting

For clarity, your output should strictly follow these rules:

- The output format should be exactly 1 integer, representing the death toll due to fighting and invasions (see subsection 1.1.3 above)
- There should be **no** whitespace characters (space, newline, tab, etc.) before or after this integer
- Output to the file specified by the second command-line parameter (not standard output)
- Do not write any other text to this file or your submission may be graded incorrectly. Use the standard output stream or standard error stream for logging if you need it. We will ignore stdout and stderr when grading.

Sample Program Execution

```
$ ./goi small_input.in death_toll.out 1
```

Sample Input File

```
10
7
7
1 1 0 0 0 2 2
1 0 0 0 0 0 2
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
3 0 0 0 0 0 4
3 3 0 0 0 4 4
2
3
0 0 0 0 3 3 0
0 0 0 0 3 3 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
6
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
9 9 0 0 0 0 0
9 9 0 0 0 0 0
```

Explanation of the Input File

This means simulate 10 iterations of a 7×7 world with the starting state

```
1 1 0 0 0 2 2
1 0 0 0 0 0 2
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
3 0 0 0 0 0 4
3 3 0 0 0 4 4
```

There will be 2 invasions. The first invasion, landing at generation 3 is

```
0 0 0 0 3 3 0
0 0 0 0 3 3 0
0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
```

The second invasion, landing at generation 6 is

```
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
9 9 0 0 0 0 0
9 9 0 0 0 0 0
```

Sample Output File

```
10
```

1.3 Starter Code

We provide a sequential implementation of Game of Invasions in the starter code. It is a (hopefully) straightforward C implementation of the rules and requirements specified above. A few implementation details to note:

- Generations (world states) are laid out in contiguous memory
- Facilities for visualizing the world states are provided, and controlled via preprocessor flags in `settings.h`
  - One of these facilities may output a file to be read by the GOI visualizer (introduced below). The file it outputs to is controlled by an optional **fourth** command-line argument when running `goi`. When we test the code, this fourth argument **will not be specified**.

The code structure is shown in Table 5.

1.3.1 GOI Visualizer

We provide a separate application, `goi_visualizer` to view your simulation. The visualizer is meant to be run on your **local machine** (because it has a graphical display). You can find the visualizer here. If updates and changes are needed, the visualizer will receive a new version number.

Using or even downloading the **visualizer is not necessary at all for completion of this assignment**. It provides a way to see your simulation come to life, and may help with debugging for small grid sizes.

For those interested, `goi_visualizer` reads in lines of JSON where each line is a world state so you could make it display anything that follows the input format.

If you experience compatibility issues or have any feedback/suggestions, email Benedict (benedictkhoo.mw@u.nus.edu).

Files/Folders	Description
`main.c`	This is the entry point of the program. It handles command-line arguments, file opening/closing, input parsing and output writing. **You should not need to modify this, though you may. If you make modifications that you want us to consider during grading (e.g. an optimization), do mention it in your report. If you fail to mention it, we will NOT consider it when grading.** That said, if you modify this file and it introduces bugs in the program (e.g. memory leaks, incorrectness, failing to handle certain cases within specification, etc.), you may be penalized for it.
`goi.c` `goi.h`	These files constitute the main logic of the simulation, and where we expect most or all of your modifications to take place.
`util.c` `util.h`	These files contain utility functions for getting and setting values from a world matrix, and printing a world state. You should not need to modify this, though you may. The same rules apply as in `main.c`
`exporter.c` `exporter.h`	These files contain the library we wrote to export world states to a format that the GOI visualizer can understand (explained in the following section). Do not modify these files. We will NOT consider any modifications in this folder when grading, even if you mention it in your report. Furthermore, you may be penalized if your modifications here reduce program correctness.
`settings.h`	This file contains two preprocessor directives for your convenience. These are toggles for printing world states out and exporting world states for the GOI visualizer, and are there to facilitate debugging. These are optional to use, and you may remove them from your submission if you do not like them. Of course, please make sure your program compiles and runs fine should you choose to remove them.
`Makefile`	Allows you to compile the sequential code provided using `make build`.
`check_zip.sh`	Script to check that your archive follows the required structure.
`sb/`	This folder contains code for a string builder library imported to implement the exporter module. You probably will not need to use this. Do not modify the contents of this folder. The same rules apply as in `exporter.c`.

Table 5: Code Structure

1.4 Your Task

Your task is to implement parallel versions of Game of Invasions using pthreads and OpenMP in C/C++. You need not use the sequential implementation given, but you are recommended to. Your parallel implementations should be bug-free, have no memory leaks and should run faster than your OpenMP implementation running one thread (assuming no artificial slowdowns) on the lab machines (otherwise there is no point parallelizing it). Apart from these programs, you will need to conduct some performance measurements and write a report. You are allowed to modify the provided sequential implementation as long as the correctness of the output result is maintained.

> Your parallel implementations should give the same result (output) as the provided sequential implementation, and execute faster on the lab machines.

1.5 Optimizing your Solution

The provided implementation is relatively simple, and thus it may seem difficult at first to identify parallelism that can be extracted by a multi-threaded model. As such, you might need to try several approaches to parallelize the implementation. In your submission, you are advised to retain your alternative implementations (even if they are slower), and explain the changes you have made in each.

You may obtain **bonus marks** by obtaining a good speedup (improvement in performance) for your **OpenMP program**. When computing your speedup, you have to compare against your OpenMP implementation running with one thread. Use the same level of compiler optimizations when computing the speedup. Avoid focusing on optimizing your pthreads implementation because that will not bring you bonus marks.

You should demonstrate your parallel implementation scales with increasing input size (**Notagain** world size) and number of threads. To analyze the improvements in performance, you should measure the execution time for carefully chosen input sizes with an increasing number of threads.

> ⚠ Distinguish any alternative implementations you include in your submission clearly from the final parallel implementations to be graded.

2 Admin Issues

2.1 Running your Programs

We provide a `Makefile` for easy compilation of the provided files. Feel free to modify this `Makefile` to include the compilation of your pthreads and OpenMP programs as well. Instead of `gcc/g++`, you may use other compilers to compile your code. You may request for installation of new tools and compilers on the lab machines. During development you might use your personal computer or the computers from the SoC Compute Cluster reserved for CS3210. Their hostnames are: `xcne3` and `xcne4`.

Your code should successfully compile and run on the lab machines. Run your program(s) with varied input sizes (world size) and number of threads used. You should select sizes that have meaningful execution times when solved by the provided sequential implementation. You might investigate further how the number of cores impacts the execution time. Use different different lab machines or vary the number of cores used for execution.

For performance measurements, run your program at least 3 times and take the shortest execution time. You should use the machines in the lab for your measurements:

- `soctf-pdc-001` - `soctf-pdc-008`: (Xeon Silver 4114)
- `soctf-pdc-009` - `soctf-pdc-016`: (Intel Core i7-7700K)
- `soctf-pdc-018` - `soctf-pdc-019`: (Dual-socket Xeon Silver 4114)
- `soctf-pdc-020` - `soctf-pdc-021`: (Intel Core i7-9700)

To view the usage of the lab machines, you can use this Telegram bot: `@cs3210_machine_bot`. Simply type `/start` to get a real time status update for all machines.

When computing the speedup of your parallel implementations, compute it against your OpenMP implementation running with one thread.

> ⚠ Avoid using the lab machines for your development. Use the lab machines only for your performance measurements once your code is working correctly.

2.2 Bonus

You may obtain up to 3 bonus marks for the following:

- **up to 2 bonus marks for speedup contest**: for achieving the best speedup among all OpenMP submissions. We will assign in total 8 bonus marks to the class, two marks for obtaining the best speedup on each type of lab machine (listed above). If an implementation tops on multiple machines only two bonus marks will be allocated to the student(s), and we will consider the next best speedup. Partial marks can be obtained for the second and third best on each machine.
- **up to 1 bonus mark for your performance analysis approach**: for using a clear, simple, yet comprehensive approach in your performance analysis. You need to analyze the execution time for carefully chosen input sizes with an increasing number of threads and hardware capabilities. Furthermore, you might have a comparison among different methods of parallelization used for this problem. Feel free to further investigate the impact on performance of different scheduling policies in OpenMP, and work distribution (chunk size). Extracting and presenting interesting insights from your measurements can be awarded bonus marks.

2.3 FAQ

Frequently asked questions (FAQ) received from students for this assignment will be answered here. The most recent questions will be added at the beginning of the file, preceded by the date label. **Check this file before asking your questions.**

If there are any questions regarding the assignment, please post on the LumiNUS forum or email Benedict (benedictkhoo.mw@u.nus.edu) or Brian (e0310531@u.nus.edu).

2.4 Submission Instructions

You are advised to work in groups of two students for this assignment (but you are allowed to work independently as well). You may discuss the assignment with others but in the case of plagiarism, both parties will be severely penalized. Cite your references or at least mention them in your report (what you referenced, where it came from, how much you referenced, etc.).

Assignment submission is due on **Mon, 27 Sep, 11am**. The grades are divided as follows:

- 3 marks – pthreads implementation (`goi_threads.c`)
- 3 marks – OpenMP implementation (`goi_omp.c`) and a `Makefile` that compiles all your implementations when calling `make build`
- 1 marks – the test cases that can be used to reproduce the results from your report
- 5 marks – a report that includes a performance comparison between the sequential and parallel implementations.

Your OpenMP/threads implementations should:

- Give the same result (output) as the provided sequential implementation
- Have no memory leaks
- Run faster than your OpenMP implementation running one thread on the lab machines. Specifically, to obtain full marks for the performance part of the implementations, both your implementations should achieve a speedup of at least 2x when running with at least 8 threads (on a machine with at least 8

cores), for a world size of 50×60 and $1,000,000$ steps. (example input `sample6.in`, but another test case will be used for grading).

Your report should include:

- A brief description of your program's design and implementation assumptions, if any.
- A brief explanation of the parallel strategy you used in your OpenMP implementation, e.g. synchronisation, work distribution, etc. If you used multiple methods to parallelize, briefly explain the reasons for choosing a specific method.
- Any special consideration or implementation detail that you consider non-trivial.
- Details on how to reproduce your results, e.g. inputs, execution time measurement, etc.
- Execution time and speedup measurements of your OpenMP implementation running one thread vs your parallel implementations. Vary input size and the number of threads, etc. State your assumptions and mention the lab machine (hostnames) that you have used in your experiments.
- In your report, present and explain your insights for at least the following things: graphs showing the execution time (y-axis) variation with input size, number of threads (x-axis) (fixed input size). You can vary the number of threads from 1 to 64. Feel free to add any other graphs or insights that you find.

> ⚠ Tips:
>
> - There could be many variables that contribute to performance, and studying every combination could be highly impractical and time-consuming. A report that investigates two or three variables sensibly, with explanations as to why these variables might affect performance (and are worth investigating) is better than a report that blindly tries every combination of variables. You will be graded more on the quality of your investigations, not so much on the quantity of things tried or even whether your hypothesis turned out to be correct.
> - Performance analysis may take longer than expected and/or run into unexpected obstacles (like your test script failing halfway). Start early and test selectively.
> - The lab machines are shared with the entire class. Please be considerate and do not hog the machines or leave bad programs running indefinitely. Again, start early or you may be contending with everyone else.

There is no minimum or maximum page length for the report. Be **comprehensive**, yet **concise**.

Submit your assignment before the deadline under LumiNUS Files. Each student must submit one zip archive named with your student number(s) (A0123456Z.zip - if you worked by yourself, or A0123456Z_A0173456T.zip - if you worked with another student) containing the following files and folders. Do not add any additional folder structure.

1. Your C/C++ code for `goi_threads.c` and `goi_omp.c` and any source or header files needed to build them. Include your improved sequential implementation, if any (`goi_seq.c`).
2. `Makefile` with a recipe named `build` that builds **all** your implementations exactly as you intend them to be graded for correctness/performance. Also remember to remove unnecessary print/export statements if you think they will affect correctness/performance. The executable names produced should be `goi_threads` and `goi_omp`. Be sure to include **everything** in your submission needed such that when `make build` is run on the lab machines, both `goi_threads` and `goi_omp` are built without issue.
3. Report in PDF format (A0123456Z_A0173456T_report.pdf or A0123456Z_report.pdf).
4. A folder, named `testcases`, containing any additional test cases (input and output) that you might have used.
5. An **optional** folder, named `scripts`, containing any additional scripts you used to measure the execution time and extract data for your report.

Once you have the zip file, you will be able to check it by doing:

```
$ chmod +x ./check_zip.sh
$ ./check_zip.sh A0123456Z_A0173456T.zip (replace with your zip file name)
```

During execution, the script prints if the checks have been successfully conducted, and which checks failed. Successfully passing the checks ensures that we can grade your assignment. You will receive 0.5% simply for having a valid submission file!

Note that for submissions made as a group, only the most recent submission (from any of the students) will be graded, and both students receive that grade. A penalty of 10% per day (out of your grade) will be applied for late submissions.