Assignment 1
Invasion of MUGLAND

CS3210 – 2023/24 Semester 1

1 Sep 2023 (updated 14 Sep 2023)

Learning Outcomes

This assignment aims to provide an introduction to parallel programming with a shared memory paradigm through 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.

This assignment specification has been amended since the original version.
The changelog can be seen at the last page.

This assignment is designed to assess your understanding on the usage of OpenMP.
Do **not** use POSIX threads and `fork()` in your solution.

1 Problem Scenario

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

MUGLAND is a two-dimensional world where life evolves from generation to generation according to a set of rules. Every so often, MUGLAND 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 continues.

1.1 Simulating the Invasion of Mugland

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.

Every cell in the grid has eight neighbors (the cells horizontally, vertically and diagonally adjacent to it). Cells along the MUGLAND world border will find that some of its neighbour live on the opposite side of the world, i.e. the grid wraps around. Figure 1.1 is an illustration of the neighbouring rule on an 8x8 grid. In this example, the neighbours of the cell labeled A (dark blue) are the cells labeled a (light blue); the neighbours of the cell labeled B (dark green) are the cells labeled b (light green); the neighbours of the cell labeled C (dark red) are the cells labeled c (light red); and the neighbours of the cell labeled D (brown) are the cells labeled d (yellow).

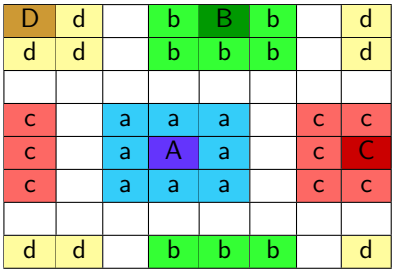


Figure 1: Examples of neighbours

To evolve from one generation, G, to the next generation, $G+1$, every cell in G will interact with its eight neighbors to determine its next state in $G+1$, according to the rules shown in Table 2.

1.1.1 Rules

Table 2 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 mutually exclusive – if none of the rules can be applied, then the cell would remain unchanged in the next generation.

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 and there is no invader about to land on this cell, it will become alive in the next generation. If multiple factions contend to reproduce in the same cell, the higher numbered faction wins.

Table 2: Rules of Invasion of MUGLAND

1.1.2 Invasions

At pre-specified world states, an invasion may occur (at most one per world state; MUGLAND 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 MUGLAND . 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, regardless of whether they would have died due to other causes.

Note that if an invader lands on a cell that would otherwise have come alive (Reproduction rule), it should not be counted.

1.1.4 Examples

This section provides examples showing the simulation rules in action. Table 3 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.)

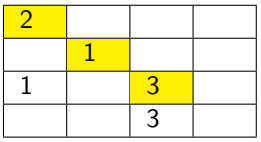
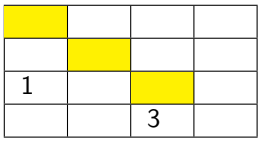
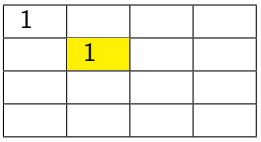
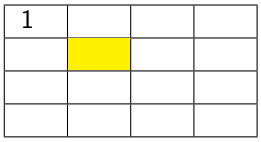
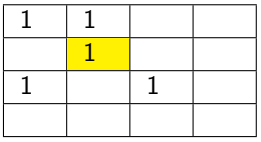
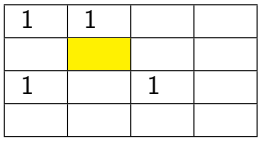
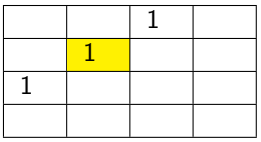
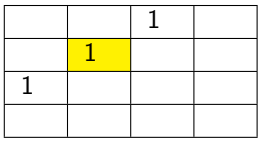
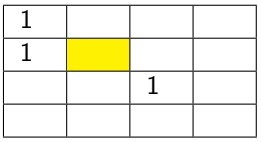
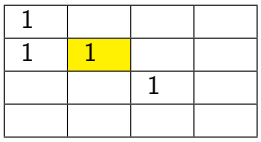
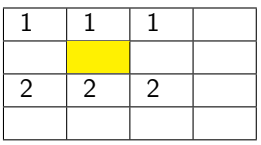
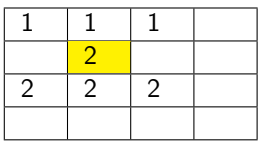
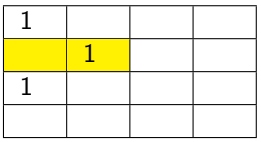
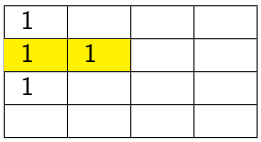
Rule	Generation G	Generation $G+1$
Fighting		
Underpopulation		
Over Population		
Survival		
Reproduction		
Reproduction Tie-breaker		
Survival + Reproduction		

Table 3: Examples of Applying the Rules

The example in Table 5 shows a simulation of a 7×10 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 4 will land.

					3	3	3
		3	3				
		3	3				

Table 4: Invasion at generation 2 in the example scenario

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 5: Example simulation of Invasion of MUGLAND

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 implementations. Varying this argument should allow running these implementations with different numbers of threads.

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. $0 \leq$ `N_GENERATIONS` $\leq 1,000,000$
- `N_ROWS`: number of rows in the 2D world. $1 \leq$ `N_ROWS` $\leq 1,000,000$
- `M_COLS`: number of columns in the 2D world. $1 \leq$ `M_COLS` $\leq 1,000,000$
- `STARTING WORLD`: the layout of generation 0 (must be `N_ROWS` $\times$ `M_COLS`). Columns are separated by spaces; rows are separated by newlines.
- `N_INVASIONS`: number of invasions in the simulation. $0 \leq$ `N_INVASIONS` $\leq 1,000$
- `N_INVASIONS` repetitions of the following (sorted in ascending order by `INVASION_TIME`):
  - `INVASION_TIME`$_i$: the generation number that the i^{th} invasion will land at. $1 \leq$ `INVASION_TIME`$_i \leq$ `N_GENERATIONS`, where each value `INVASION_TIME`$_i$ is distinct
  - `INVASION_PLAN`$_i$: basically a world layout (`N_ROWS` $\times$ `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

```
$ ./iom 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

```
12
```

1.3 Starter Code

We provide a skeleton code which can be used as a starting point for your OpenMP implementation. The code structure is shown in Table 6.

Files/Folders	Description
`main.cpp`	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.
`iom.cpp` `iom.h`	These files constitute the main logic of the simulation, and where we expect most or all of your modifications to take place.
`Makefile`	Allows you to compile your implementation using `make build`.
`run_slurm.sh` `slurm_job.sh`	Shell script provided to run your files. `run_slurm.sh` is the main entry point for the shell script, which will build and execute the program you have made. `slurm_job.sh` specifies the job that will be submitted to Slurm. You may or may not use this script to submit your own program to the Slurm queue.

Table 6: Code Structure

In addition, we also include a sequential implementation (only an executable called **iom_sequential** is provided). This implementation will only be used for correctness reference. We have tested our sequential implementation but as always, it is possibly incorrect. If you notice any issues, please do let us know. Our contact details are available in Section 2.3.

1.4 Your Task

Your work is to implement parallel versions of Invasion of MUGLAND using OpenMP in C/C++. The following are the requirements for your parallel implementations:

- **Correctness** Your implementation needs to be correct, i.e. have the same output as the sequential implementation executable given.
- **Performance** On the lab machines, your parallel implementation running on multiple threads should be faster than it running on one thread.
- **Others** Your implementation should be free of memory leaks.

1.5 Optimizing your Solution

Your program needs to have a minimum speedup of 2x running on 8 threads in the lab machines (for details, see Section 2.4). When computing your speedup, you have to compare against your own OpenMP implementation running with one thread. Use the same level of compiler optimizations when computing the speedup. In addition, note that you may obtain **bonus marks** from the contest based on your program's execution time (Section 2.2)

You should demonstrate your parallel implementation scales with increasing input size (MUGLAND 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 suit your needs. Your code needs to be compilable with **both** g++ and clang++. You may request for installation of new tools and compilers on the lab machines. During development you might use your personal computer prior to sending your job via Slurm.

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 your implementation on one thread. You might investigate further how the number of cores impacts the execution time. Use different lab machines or vary the number of cores used for execution.

For ease of running your jobs on Slurm, we are providing the scripts named `run_slurm.sh` and `slurm_job.sh`. You may use these scripts as you see fit.

For performance measurements, run your program *at least 3 times on Slurm* and take the **average** execution time **using perf**. You should use the machines in the lab for your measurements:

- `soctf-pdc-004` - `soctf-pdc-008`: (Xeon Silver 4114)
- `soctf-pdc-012` - `soctf-pdc-016`: (Intel Core i7-7700)
- `soctf-pdc-018` - `soctf-pdc-019`: (Dual-socket Xeon Silver 4114)
- `soctf-pdc-020` - `soctf-pdc-021`: (Intel Core i7-9700)
- `soctf-pdc-023` - `soctf-pdc-024`: (Intel Xeon W-2245)

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

Avoid hogging the lab machines. Use the lab machines only for your performance measurements once your code is correct, and you just need to make it faster. Excessive usage of computing resources will decrease your job's priority in Slurm. However, keep in mind that many students will be submitting Slurm jobs closer to the deadline, so **test early and often!**

Use Slurm to run your performance measurements. Do not run performance measurements on the login nodes as your performance measurement might be impacted by other users.

2.2 Bonus

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

- **up to 2 bonus marks for speed contest**: please refer to https://nus-cs3210.github.io/student-guide/a1-bonus/ for details.
- **up to 1 bonus mark for your performance analysis approach**: for using a clear, simple, yet comprehensive approach in your performance analyses. You need to analyze the execution time plus one other metric 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). Additionally, you may also try an analysis on different machine architectures (instructions on how to access the AMD machines are in https://nus-cs3210.github.io/student-guide/a1-bonus-soc-slurm/) 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 Canvas forum or email Theodore (theodoreleebrant@u.nus.edu).

2.4 Submission Details

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 **27 September 2023, 2 pm**. The grades are divided as follows:

- 4 marks – OpenMP implementation and a `Makefile` that compiles all your implementations when calling `make build`
- 6 marks – a report that includes a performance comparison between the sequential and parallel implementations.

Your implementation should:

- Give the same result (output) as the provided sequential implementation.
- Have no memory leaks.
- Run faster than your OpenMP implementation running on 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. Speedup in this context refers to the wall-clock time speedup.

Your report should have:

- A maximum of 4 A4 pages (2 sheets of paper) for main content, excluding appendix.
- A reasonably-sized font (comparable to 11-point Arial) and page margins (at least 1 inch on each side).
- Have visually distinct headers for each content items below:
  - (2.5 marks) Description on:
    - your program's design and implementation assumptions, if any.
    - the parallel strategy you used in your implementation, e.g. synchronisation, work distribution, etc. If you used multiple methods to parallelize, briefly explain the reasons for choosing a specific method.
    - Implementation details or special considerations that you consider non-trivial.
  - (2.5 marks) Description, visualization, and data on your execution, including:
    - how to reproduce your results, e.g. inputs, execution time measurement, etc. State your assumptions and mention the lab machines that you have used in your experiments.
    - the performance (execution time and one other metric) as well as speedup measurement of your implementation on one thread vs multiple threads, given varying input sizes (e.g. number of rows, number of columns, number of generations, etc.) and varying number of threads. Note that `perf` needs to be used as part of your performance measurement.
    - an explanation on which aspect of the input size affects speedup the most for your implementation and why you think this is the case.
    - any other insights that you find.
  - (1 mark) A description of ONE performance optimization you attempted (if any) with analysis and supporting performance measurements. Include at least one summary graph in your report and link to your supporting measurement data (for example, a .csv file in the repository)

> Your report should be self-contained. If you write part of your report somewhere else and reference that in your submitted "report", we reserve the right to ignore any content outside the submitted document. An exception is referencing a document containing measurement data that you created as part of the assignment – we encourage you to do this.

If headers, spacing, or diagrams cause your report to *slightly* exceed the page limit, that's ok – we prefer well-organized, easily readable reports.

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 length for the report. Be **comprehensive**, yet **concise**.

2.5 Submission Instructions

The submission for this assignment will be through GitHub classroom and Canvas Quiz.

- **Canvas Quiz Assignment 1**: Take the quiz and provide the name of your repository and the commit hash corresponding to the `a1-submission` tag; if you are working in a team, only one team member needs to submit the quiz. If both of you submit, we will take the latest submission.
- **GitHub Classroom**: The implementation and report must be submitted through your Github Classroom repository. The GitHub Classroom for Assignment 1 can be accessed through https://classroom.github.com/a/-QsEqzNH.

The GitHub Classroom submission must adhere to the following guidelines:

- Name your team `a1-e0123456` (if you work by yourself) or `a1-e0123456Z-e0173456` (if you work with another student) – substitute your NUSNET number accordingly.
- Push your **code and report** to your team's GitHub Classroom repository.
- Your report must be a PDF named `<teamname>.pdf`. For example, `a1-e0123456-e0173456.pdf`. `<teamname>` should **exactly match** your team's name; if you are working in a pair, please DO NOT flip the order of your admin numbers.
- Tag the commit that you want us to grade with `a1-submission`; if no such tag exists, we will use the most recent commit.

A penalty of 5% per day (out of your grade for this assignment) will be applied for late submissions.

Final check: Ensure that you have submitted to both **Canvas** and **GitHub Classroom**. Canvas should contain your commit hash and repository name, and GitHub Classroom should contain your code and report.

Changelog

14 Sep: Output of Sample Program Execution is corrected, highlighted in pink