Assignment 1
Data and Task Parallelism Using Pthreads
Due date: Sunday, October 9, 11:59pm

The purpose of this assignment is to familiarize yourselves with using Pthreads to solve parallel problems. You will write short programs that will use data parallelism and task parallelism to create the solutions.

Both programs will parallelize the classic computer science problem: Conway's Game of Life. This "game" is actually a self-contained simulation with simple rules, which uses a 2D grid. Details can be found here: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

This assignment is individual work and is subject to the University Academic Misconduct Policy. See course outline for details.

Data Parallelism

This algorithm will require the threads to read overlapping squares to calculate the new state of the grid, but it does not require overlapping writes. You will need two arrays to act as the grid for the game. One array is read-from and the other one is written-to.

Each location in the grid should be written to by a single thread. This does not mean that you need to have one thread per location - the number of threads is provided by the user. Just make sure that you never try to modify the same location by more than one thread - and thus create a race condition.

The program is run on the command line and uses parameters to configure the size of the game and the number of threads.

The usage for the program is:

```
gol_data nThreads gridSize nIterations -d
```

where:

- `nThreads` is the number of threads the program will create to run the game. This is a required parameter.
- `gridSize` is one number that represents both the height and width of the grid. This is a required parameter.
- `nIterations` is how many iterations of the game the program will run before exiting. This is a required parameter.
- `-d` indicates that the program should print out the grid after each iteration. This is an optional argument:
  - If the `-d` is present then the output should be displayed
  - If the `-d` is not present then the output is not displayed. Each new grid is calculated, but the grid is never drawn on the screen. This is an optional parameter.

The parameters will always appear in this order. Make sure you do some basic error checking of your command-line arguments.

For example:

- `gol_data 4 10 100` will create 4 threads, a grid size of 10x10, run for 100 iterations, and never display the grid.

- `gol_data 2 100 5 -d` will create 2 threads, a grid of 100x100, run for 5 iterations, and will display the grid after each iterations (5 times).

All created threads should be used to calculate a proportional amount of the grid. For example, if there are two threads, then each one should calculate half of the grid. If there are three threads, then each one should calculate one third of the grid. Try to keep the chunks relatively even - you do not want one thread doing way more or way less work than the rest, since that would affect the efficiency of your code.

Randomly initialize the starting grid with a reasonable number of populated squares. This program will require two arrays for the grid. One which is the current grid and one which is used to store the newly calculated grid.

Name your C program `gol_data.c` and the executable `gol_data`.

Task Parallelism
This program also implements the Game of Life, but instead of having all threads perform the same actions, each thread specializes in a different operation.

One thread only counts the number of neighbours for each square in the grid. The location for each square is stored in a queue that indicates if the square should be occupied in the next iteration. There will need to be two queues:
- one indicating that the location should be occupied (live) for the next iteration (*live* queue)
- another indicating the location should be empty for the next iteration (*empty* queue)

For example, in the live queue you could have something like `(10, 10)`, which means in the next iteration the location 10, 10 should be occupied. You can encode the position on the board any way you wish. It does not need to be two integers.

The second thread should read through the *live* queue and update the next iteration game board. This should mark the location on the board as occupied.

The third thread thread should read through the *empty* queue and update the next iteration game board. This should mark the location on the board as unoccupied.

For each iteration of the game, every square on the board will be placed in one of the two queues.

Once an element in the occupied or unoccupied queue has been processed and the board has been updated that item in the queue must be removed. You can implement the queue any way you wish, but each iterations should involve every square on the game board being placed in one of the queues (by the first thread).

All three threads must run simultaneously. Thread one will be filling the two queues while threads two and three are removing elements from the queues. Do not have thread one completely fill the queues and then start threads two and three to empty them. The only time you might wish to restrict the queues from running is when you swap from the old board to the new board after one complete iteration of the game.

The threads communicate through queues which must be locked when they are being written. You can choose the method of locking the queues. However, mutexes with condition variables are a natural fit here.

Name your C program `gol_task.c` and the executable `gol_task`. The usage for the program is:

```
gol_task gridSize nIterations -d
```

The arguments are the same as for `gol_data`, expect the number of threads is hard-coded for the task-parallel solution.

Report

In addition to writing the solution code, you will need to submit a report comparing the performance of the data-parallel and task-parallel solutions. You will need to do the following:

- Run `cat /proc/cpuinfo | grep processor | wc -l` in your Docker container to see how many CPU cores Docker can use.
- Run `gol_data` with 1, 2, 3, and 4 threads. For each number of threads, repeat the execution at least 10 times. Record the run time for each individual run - you can borrow the code from my examples. You will end up with at least 40 run times - 10 per each number of threads
- Run `gol_task` the same number of times - i.e. at least 10. Again, record all the run times. This will give you 10 more data points.
- Do all the runs sequentially, during the same session, so you are comparing the performance under reasonably similar load conditions on your machine.
- For all the runs, use the following parameters:
  - `gridSize` = 100 x 100
  - `nIterations` = 10000
  - We do not want the board to be displayed in these runs, so **do not** run the code with the `-d` option.

Display your results using a box-and-whisker plot (sometimes known as just a box plot): https://en.wikipedia.org/wiki/Box_plot. You can do this in Excel or in R. You'll have 5 box plots in total, together on the same image: 4 representing `gol_data` with various numbers of threads, and one representing `gol_task`. Give us your analysis of the results - what your expected in terms of performance, what your have observed, and why that might be. When performing the analysis, keep in mind the max number of cores that are available to Docker.

The report must be submitted as a PDF file called `A1report.pdf`.

Submission and Evaluation

Submit the assignment using Moodle. Submit only the source code, report, and the makefile. Bundle everything in an archive.

The assignment will be marked using the standard CIS*3090 Docker image provided on the course website and discussed in class. We will download and run a fresh image, create container, and use Docker to run and grade your code. Make sure you test your code accordingly.

The TA will unpack your code and type "`make`". They will then try to run executables named `gol_data` and `gol_task`. If the makefile is missing, the make command or executing the programs do not work then you will lose marks. As a result, it is always a good idea to unpack and test the file you are submitting to be sure that what you submit actually compiles.