The aim of this programming assignment is to simulate Conway's Game of Life (GoL) on a square grid and parallelize your code using OpenMP. The GoL is played on a square grid with wrap-around connections. The grid is simulated by a two-dimensional array with N rows and N columns. Each cell in the array has four neighbours, north, south (vertically above and below), east and west (horizontally to the right or left). The wrap-around connection implies that the last cell of a row has the first cell of the row as its eastern neighbour. Similarly the first cell of a row has the last cell of the row as its western neighbour. Similar wrap-around connections can be defined for the first and last cells of a column.

There are two kinds of cells in a GoL simulation, *dead* and *live*. These can be simulated easily by storing 0 or 1 in the locations of a two-dimensional array. The GoL proceeds by evolving the cells according to some simple rules. Whereas one can devise any rules, the usual rules are the following.

A live cell dies if it has less than two live neighbours due to loneliness;

A live cell with four live neighbours die due to overpopulation;

A live cell with two or three live neigbours continues to live;

Any dead cell with exactly three live neighbours becomes live due to reproduction;

The game starts with some cells that have been initialized as live, and then the system continues to evolve according to the rules mentioned above. You can read more about the Game of Life from this Wikipedia article, or this Wolfram MathWorld article. I have checked both of these articles for authenticity and correctness of information.

Tasks: Your task is to simulate the game of life using OpenMP and study the speed-up. You should simulate using arrays of size 128X128, 256X256, 512X512, 1024X1024 and 2048X2048 using three of the initial patterns mentioned in the articles linked above. Please note that the original GoL assumes a 8-neighborhood, and we are using a 4-neighborhood (also known as the von Neumann neighborhood). You will not get the patterns that are shown in those articles. As such, any random initialization of the live cells is fine, we are interested in the speed-up rather than in the actual patterns. Each simulation should be run for 100 time steps, meaning that states of all the cells in the array should be updated at least 100 times for timing analysis. You should compare the sequential times and the times obtained by OpenMP, and plot a graph comparing these execution times. You should also plot the complete array for at least three time steps by using the colour black for live cells and the color white for the dead cells. This is required for me to check that your simulation is correct. The first plot should be the initial locations of the live cells. At least two other plots should show the evolution at two other time steps that you should mention clearly.

All plots can be made by Matlab which is available from the lab computers, or by establishing a VPN connection to the machines in CSSE. There are also excellent python packages for plotting that you may consider. Here is a tutorial for plotting in Matlab.

You need to do the following for plotting a simulation. First, write the contents of the array in a two-dimensional form, each 0 or 1 should be separated by space. This can be done by using `fscanf` in C. Next use the following two Matlab commands:

```
M=dlmread('filename')
imagesc(M)
```

`filename` is the name of the file where you store your file (notice the single quotes). The first

command will load your file in the two-dimensional Matlab array M. The second command then plots it as a two-dimensional picture with different colours for 1s and 0s. It is possible to make the pictures nicer and save as pdf files.

Marking: The marking will be done following this scheme: 8 marks for correct implementation and appropriate use of OpenMP directives. 4 marks for the report, and 3 marks for any additional experiments like using different number of threads, experiments in parallelization in different ways etc.

Allocating large arrays in C: You may not be able to allocate large arrays in stack (you can listen to the video on stack and heap if you do not understand this). You can allocate the storage for an array in the heap in the following way:

Suppose the array is `int f[NSUB+1];`

You have to declare a pointer to double:

```
int *f;
```

and then allocate storage inside your function:

```
f=(int *)malloc(sizeof(int)*(NSUB+1));
```

This can be extended to declare a 2-d array in the heap.