Project Report

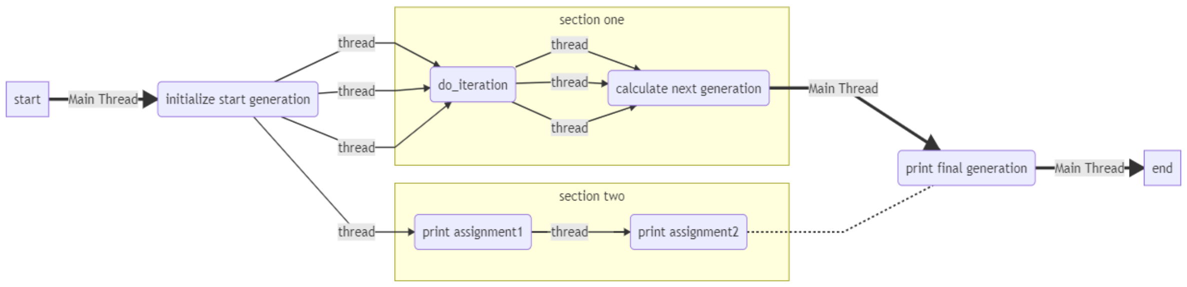
Team: CTZL

Report submitted: Feb.17th, 2021

Member: Zhiyong Liu, Ran Tao, Yichao Zhou

General Structure:

Firstly, we set the max threads of this project equals to the logical cores of the machine. In addition, using max threads initialize the start generation with random status. The next step is to separate one thread to do the print problem others execute the iteration problem.
As for the iteration strategy, multiple threads were used to do the iteration calculate the do_iteration function. Inside this function, multiple threads were used to calculate the next status of the different cells simultaneously.



Parallelization choices:

In the main function, the program starts with the initialization of grid with user specified size and iteration steps. We parallelized this initialization process (which is a simple for loop) and made it save some time when creating large grids but since it is only done once at the beginning, if the iteration steps goes up, the significance of this optimization becomes less and less noticeable.

It then comes the main parallelized part in our program, in which we split the iteration process and printing process into two separate sections. The printing part is given a single thread because we believed writing to files cannot be parallelized as it has to be ordered. We gave the iteration section all the rest threads (*MAX_THREADS-1)* as the iteration process can be further parallelized.

In the section one where all iterations go on, we designed all the rest threads (*MAX_THREADS-1*) to parallelize the iterations. The *MAX_STEPS* times of iteration are parallelized by multiple threads. To protect the data from being modified by multiple threads at the same time, we set the critical section for the code block which accesses the grid data. Therefore, even if a later iteration depends on the result of a previous iteration, iterations by multiple threads can finally produce accurate result. For example, if n iterations are completed by the threads at one time, then current stage of grid is n iterations from the initial state, which is not related to the value of indexes completed. Within each iteration, two layers of nested loops are used to calculate new grid. To accelerate this procedure, we used the parallel for and collapse (2) to make it collapsed into one large iteration. Then, this for block can be parallelized and makes each iteration faster.

In section two in which all printing carries on, it starts with printing a saved initial grid called *my_grid_start*. So, in this process, the other threads can keep on the iteration without changing what is getting printed. We designed our program to produce three output images (one for the initial grid, one for the middle of all iteration, the other one for the final grid after all iterations). So after printing the first one, we used an empty while loop to wait for the iteration number to reach the one we want (half of iteration steps). This iteration number gets updated in the other section after one time of iteration gets completed. When that iteration number has been reached, the printing section will start the printing process, while the other section for iteration will also keep on going.

After all iterations have been finished in section one, the parallelized regions join and perform the printing for the final state of the grid. We believed that this process cannot be optimized since we have to wait for all iterations done in order to have that grid ready to print.

Performance Test and analyze:
Within 100 Iteration steps and 4 logical cores calculation, when the grid size increases more, accelerating of parallel programming becomes obvious.

	100*100	**1000*1000**	**10000*10000**
Parallel time (s)	0.120848	6.7809	648.167
Serial time (s)	0.123	9.058	841.233

Within 100 Iteration steps and 1000*1000 grid, when increase the logical cores the calculation time will decrease.

	serial	**2 logical cores**	**3 logical cores**	**4 logical cores**
Time (s)	9.058	7.84829	7.34142	6.7809

GitHub repo Link:
https://github.com/acse-2020/group-project-ctzl

Breakdown of team CTZL:
Zhiyong Liu: Parallelisation strategy (clion(g++-10), openMP 2.0, Mac OS with 8 logical cores)
Ran Tao: Printing strategy (Visual Studio 2019, openMP 2.0, Windows with 4 logical cores)
Yichao Zhou: Printing strategy, report (Clang++, openMP 5.0, Mac OS with 4 logical cores)