Design and Implementation Assumptions:

Parallel Strategy:
We parallelize the generation of new world states. This is achieved by using the "#pragma omp parallel for" directive to parallelize the outer loop that iterates over the rows of the grid. This means that the computation of the new state for each row is done in parallel, as long as there are enough threads available.

In particular, our parallel begin and parallel end pattern follows the pattern of SPMD. This is due to the fact that a different set of operations are performed depending on whether the cell is dead or alive (for dead cells, we only check for reproduction and invasion). This reflects the SPMD pattern.

The type of parallelism is Data Parallelism because the same operations are being applied to each different element in the matrix. In the calculation of the new state, each cell can be calculated independently since the only dependency is the previous state which each cell only reads from.

Synchronization is achieved by the implicit barrier at the end of the parallel region, which ensures that all threads have finished their work before proceeding to the next generation. Moreover, the atomic operation "#pragma omp atomic" is used to update the death count, ensuring that this shared resource is accessed by only one thread at a time to avoid race conditions

To synchronize the accumulation of the death toll we decided to use the reduction(+:deaths) clause. This ensures that each thread gets its own private copy of deaths and all these private copies are combined (added in this case) at the end of the parallel region. The intention is to reduce contention for the deaths variable.

The work distribution in this implementation is mainly static, as each thread is assigned a roughly equal number of rows to compute. However, the actual workload will vary depending on the state of the cells in each row.

Implementation Details:
Our code checks for reproduction, fighting, underpopulation, overpopulation, survival, and invasions. These checks are implemented in a nested loop structure, checking each cell in the grid and its eight neighbors to determine its new state.

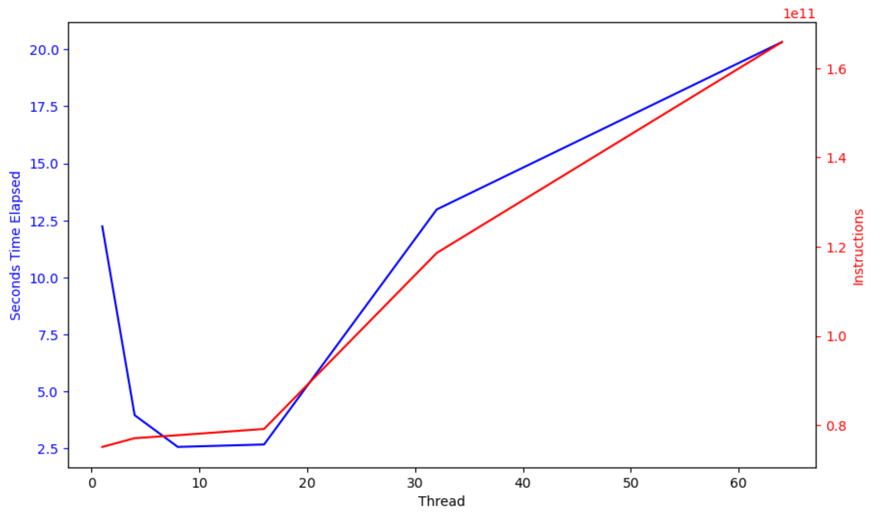
The code also includes a mechanism to handle invasions. If an invasion is scheduled for the current generation, it will check each cell to see if it is being invaded and adjust its state accordingly. This is done after the other rules have been applied, as specified in the assignment.

Performance Analysis

We first looked at the speedup provided by multi-threading. We generated a random test case with the following configuration:

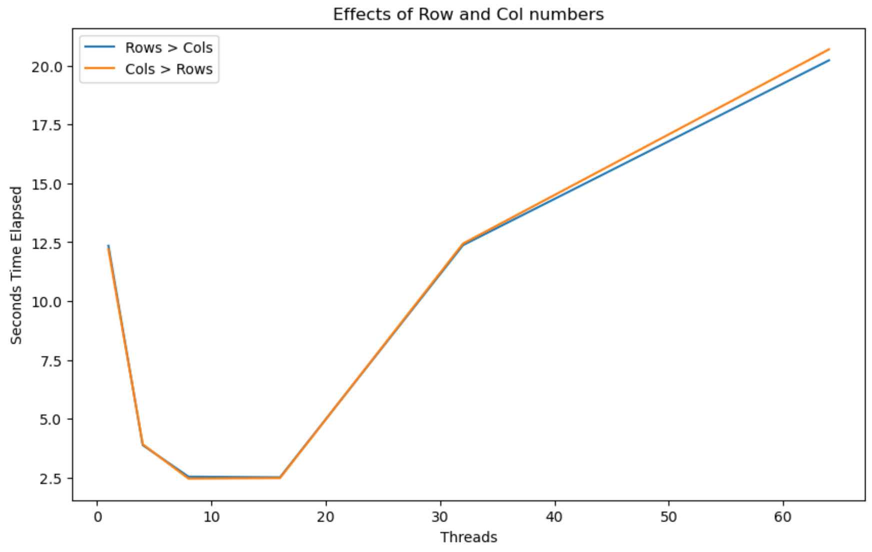
nGeneration	Row	Col	nInvasion
100000	50	60	200

We ran our program on a Xeon Silver 4114 with 1, 4, 8, 16, 32 and 64 as the input for nthreads. The results are shown below:



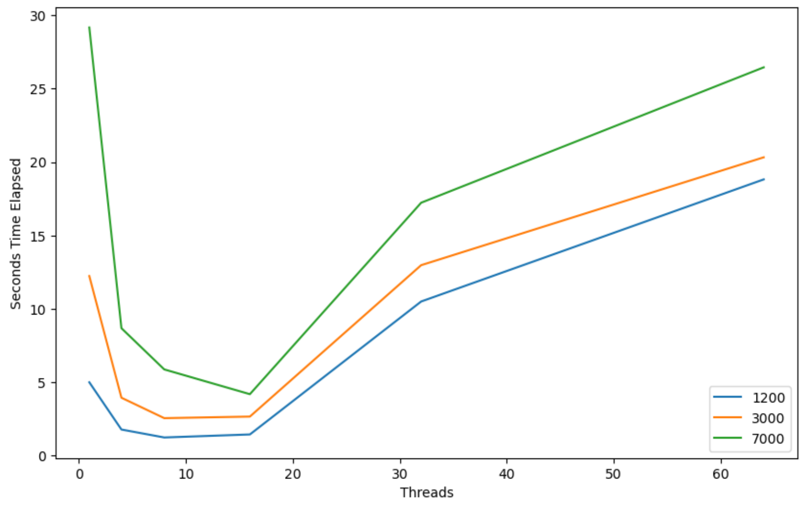
The diagram consists of two plots, one for the time taken and one for the number of instructions executed. With reference to the blue line, we can observe an initial speed up. This was expected due to the performance gains from parallelism. The speed up is followed by a plateau and a slow down which occurs at some point between 16 and 32 threads. At approximately the same point, we can also observe with the red line that the number of instructions executed begins to increase significantly. Since the test case ran was constant, the increase in instructions can be attributed to the overheads, like context switching, caused by allocating more threads. This result was also expected as the Xeon Silver 4114 has 20 hardware threads, which explains why the slow down occurred between 16 and 32 threads.

We also investigated the effects of the row to column ratio to find out if the ratio affected cache access patterns and thus performance. We generated two test cases, one with a 30x100 matrix and another with a 100x30 matrix. The results are shown below:



We concluded that the row to column ratio did not play a significant role in affecting performance.

Next we investigated the effect of matrix size on performance. The results are shown below:



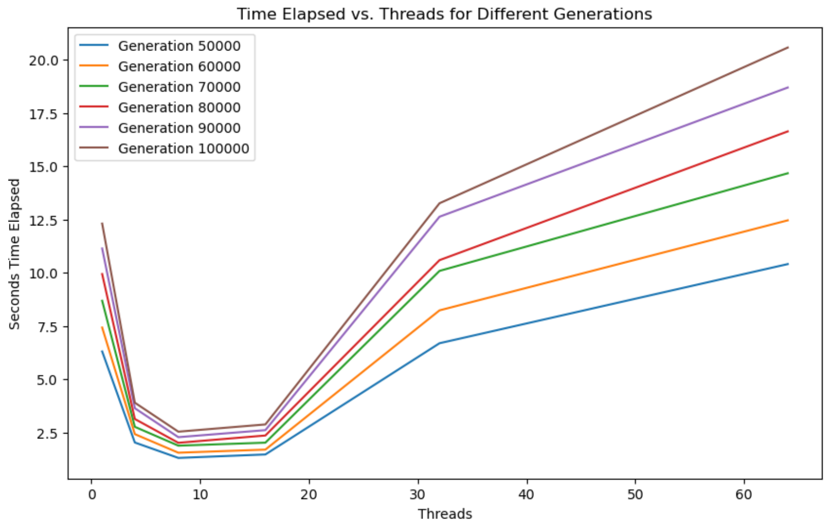
# of cells	1200	3000	7000
Max measured speed up per Matrix Size	4.06x	4.80x	6.80x

The legend on the bottom right shows the number of cells in the test case. We can observe that the maximum speed up measured increases with the input size of the matrix.

Our implementation parallelized the calculation of each row of the matrix. When the matrix size is small, the sequential part of the program (which includes setting up the parallel environment, dividing the tasks among the threads and creating a matrix copy) takes up a larger proportion of the total execution time. Moreover, parallel computation has an overhead associated with it. This overhead includes the time taken to spawn threads, distribute tasks, synchronize threads, and collect results from all threads. When the matrix size is small, this overhead can be a significant portion of the total computation time.

However, as the matrix size increases, the parallelizable part of the program (which includes the computation of cell states) becomes dominant and overhead does not increase proportionally. This means that the efficiency of parallelization improves.

Next we investigated the effect of the number of generations on performance. The results are shown below:



# of generations	50000	60000	70000	80000	90000	100000
Max measured speed up per Generations	4.85x	4.79x	4.62x	4.95x	4.86x	4.85x

We can observe that the maximum speed up measured stays roughly constant with an increase in the number of generations. This is because our implementation does not parallelize the calculation of each generation. As such, the speed up gained from parallelization is rather constant since the matrix size was fixed in this experiment.

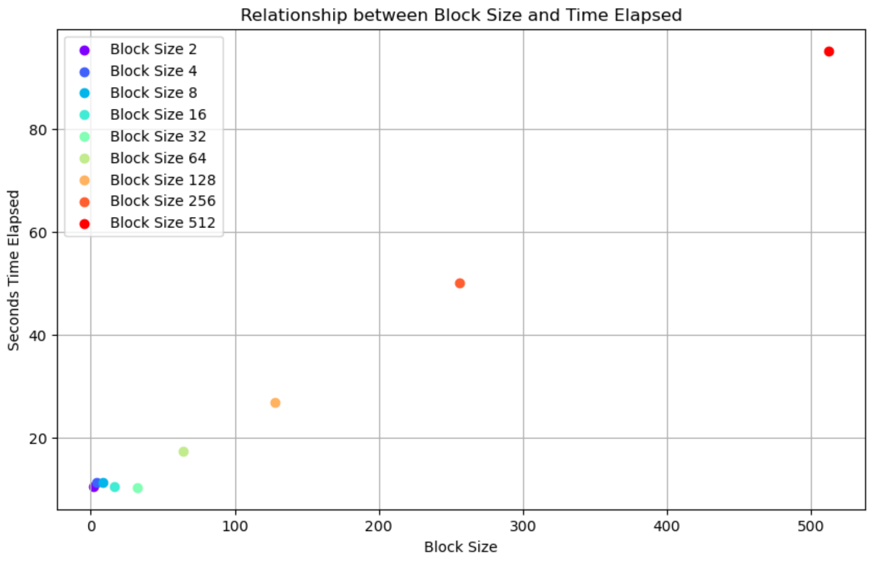
With the above analysis, we found that the input variable that most significantly affects speed up is the matrix size. The full explanation is written above in the section investigating the effect of matrix size on performance.

(1 mark) A description of ONE performance optimization you attempted (if any) with analysis and supporting performance measurements.

We attempted tiling by subdividing the matrix into smaller submatrices and having one openMP thread handle each submatrix. We measured the time elapsed using a test case with the following configuration:

nGeneration	Row	Col	nInvasion	nThreads
5000	512	512	200	40

The runs were all done on the Dual-socket Xeon Silver 4114 with the results shown below:



Our implementation showed that block sizes ranging from 4x4 to 32x32 did not cause any significant slow downs compared to our original implementation. However, beyond 32x32, we observed a considerable decrease in performance. We investigated the L1 cache loads and misses and found that they were largely similar across all runs, indicating that they were not the cause of the slow down. We believe that the slow down beyond 32x32 is due to the reduced parallelism resulting from the larger block size. For instance, at 512x512, a single thread is handling the entire matrix. This also explains the linear relationship we see in the graph. Based on our findings, we concluded that tiling did not improve cache performance the way we expected it to, and thus did not lead to any overall performance improvements.

Appendix:

We generated one random test case per configuration for our performance analysis. The configurations are shown below. We attempted to run each row on each different machine.

nGeneration	Row	Col	Cells	nInvasion	nThreads
100,000	50	60	3000	200	4
100,000	50	60	3000	200	8
100,000	50	60	3000	200	16
100,000	50	60	3000	200	32
100,000	50	60	3000	200	64
100,000	80	90	7200	200	4
100,000	80	90	7200	200	8
100,000	80	90	7200	200	16
100,000	80	90	7200	200	32
100,000	80	90	7200	200	64
100,000	30	40	1200	200	4
100,000	30	40	1200	200	8
100,000	30	40	1200	200	16
100,000	30	40	1200	200	32
100,000	30	40	1200	200	64
100,000	100	30	3000	200	4
100,000	100	30	3000	200	8
100,000	100	30	3000	200	16
100,000	100	30	3000	200	32
100,000	100	30	3000	200	64
100,000	30	100	3000	200	8
100,000	30	100	3000	200	16
100,000	30	100	3000	200	32
100,000	30	100	3000	200	64

100,000	50	60	3000	300	4
100,000	50	60	3000	300	8
100,000	50	60	3000	300	16
100,000	50	60	3000	300	32
100,000	50	60	3000	300	64
100,000	50	60	3000	400	4
100,000	50	60	3000	400	8
100,000	50	60	3000	400	16
100,000	50	60	3000	400	32
100,000	50	60	3000	400	64
50,000	50	60	3000	400	4
50,000	50	60	3000	400	8
50,000	50	60	3000	400	16
50,000	50	60	3000	400	32
50,000	50	60	3000	400	64
60,000	50	60	3000	400	4
60,000	50	60	3000	400	8
60,000	50	60	3000	400	16
60,000	50	60	3000	400	32
60,000	50	60	3000	400	64
70,000	50	60	3000	400	4
70,000	50	60	3000	400	8
70,000	50	60	3000	400	16
70,000	50	60	3000	400	32
70,000	50	60	3000	400	64
80,000	50	60	3000	400	4
80,000	50	60	3000	400	8

80,000	50	60	3000	400	16
80,000	50	60	3000	400	32
80,000	50	60	3000	400	64
90,000	50	60	3000	400	4
90,000	50	60	3000	400	8
90,000	50	60	3000	400	16
90,000	50	60	3000	400	32
90,000	50	60	3000	400	64
100,000	50	60	3000	400	4
100,000	50	60	3000	400	8
100,000	50	60	3000	400	16
100,000	50	60	3000	400	32
100,000	50	60	3000	400	64
100,000	50	60	3000	400	8
100,000	50	60	3000	800	8
100,000	50	60	3000	1200	8
100,000	50	60	3000	1600	8
100,000	50	60	3000	2000	8
100,000	50	60	3000	2400	8