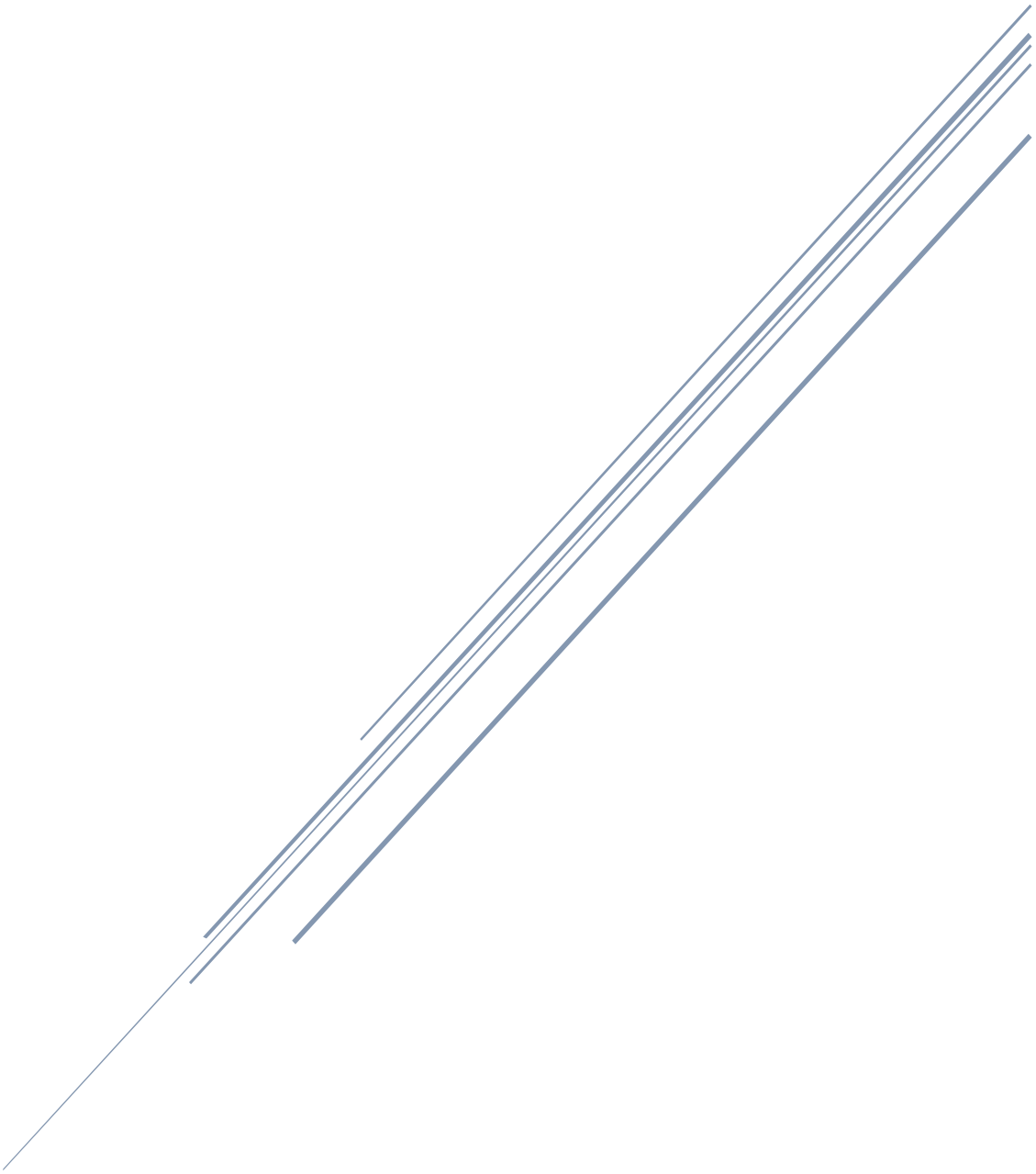
CITS3402 High Performance Computing

2018 Assignment 1 Report

Minrui Lu, 22278526 Laura Peh, 21986167



1. Running the program:

- Compilation: gcc –fopenmp -std=c11 -o assignment1 assignment1.c
- Execution format: assignment1 [width] [probability] [num_iterations]
- Execution example: assignment1 2048 0.5 100
- For all tests in this report:
    - Sizes tested: 128x128, 256x256, 512x512, 1024x1024, 2048x2048
    - Assuming a random canvas initialization, Probability: 0.5 (and 0.3, 0.7 for the plots)
        - The probability means the proportion of live cells in the square grid.
    - Number of iterations: 100
    - Computer specifications : Maths Computer Lab computers: 4 cores, 8 threads, Intel i7-4790S, CPU 3.20 GHz

2. Introduction:

In this assignment, Conway's Game of Life (4-neighborhood version) was implemented in parallel using OpenMP and multiple threads on a computer. The purposes were to compare a parallel implementation with a sequential implementation and by parallelization to achieve a better performance. In the Game of Life, there is a wrap-around square grid (matrix) in which each cell represents a life. Dead and live statuses are stored as 0 and 1 in the matrix respectively. A live cell with 0, 1 or 4 neighbors dies, but remains alive with 2 or 3 neighbors. For a dead cell, if it has 3 live neighbors, it becomes alive.

To test the speed up in the Game of Life, the times to run a sequential code execution and parallelized code execution were calculated. Moreover, the time intervals of code execution with parallelization at different parts of the program were tested. Lastly, the program performance with different threads allocated was also compared.

3. Methodology:

Each experiment was timed ten times to calculate an average. These averages are presented in the results in Section 4. A discussion then follows as a brief explanation for the results, and to conclude which implementation is the fastest.

3.1. Comparison of Parallelization at Different Sections:

The Game of Life implementation can be parallelised effectively in the presence of for loops. For loops are present in three places in the code - creating the canvas (allocating memory for an empty grid array), initializing the canvas, and calculating the next generation. The corresponding functions in project1.c are create_canvas(), initialize_canvas(), and next_generation(). Combinations of all three parallelizations were tested, on a 2048x2048 grid. The fastest was found using only parallelization of the next generation function, and not the others.

3.2. Comparison between Sequential and OpenMP:

Since the only significant speed up was in the next generation parallelization, the other tests did not involve the parallelization of memory allocation or initialization of the canvas. Instead, the time performance of only three types of implementations were compared: (1) sequential, with no OpenMP directives; (2) parallelization on every cell; and (3) parallelization on every row. The two types of parallelization here were chosen to compare the difference between the implementations. These tests were carried out on square grids of size 128, 256, 512, 1024 and 2048.

3.3. Comparison between Different Thread Numbers:

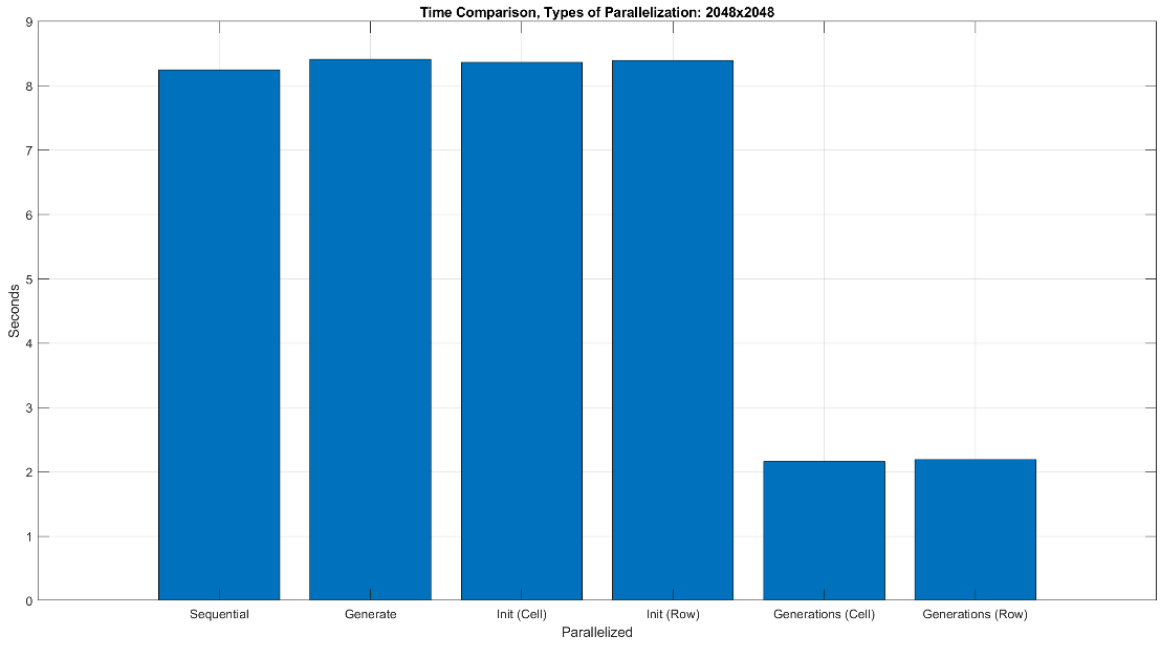
Lastly, implementations using different numbers of threads, from 1 to 12, were tested, using a 2048x2048 size grid with a parallelization using the cell implementation.

4. Results:

4.1. Comparison of Parallelization at Different Sections:

The average execution times for a size 2048 × 2048 grid are:

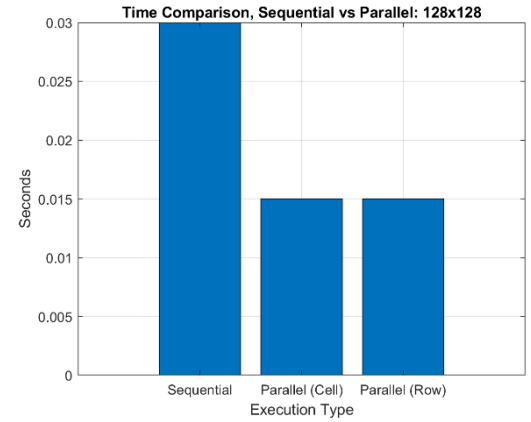
Sequential	8.25s
Parallelization in the create_canvas() function	8.414s
Parallelization for each cell in the initialize_canvas() function	8.3679s
Parallelization for each row in the initialize_canvas() function	8.3939s
Parallelization for each cell in the next_generation() function	2.164s
Parallelization for each row in the next_generation() function	2.191s

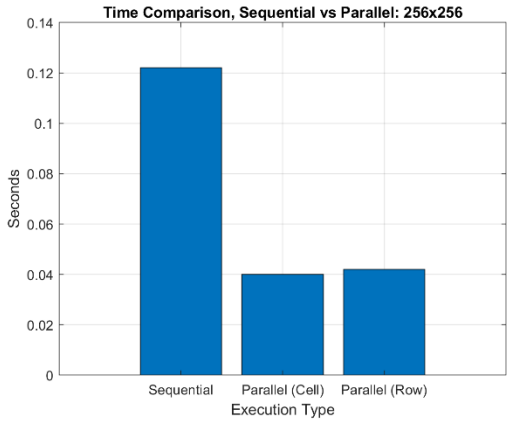


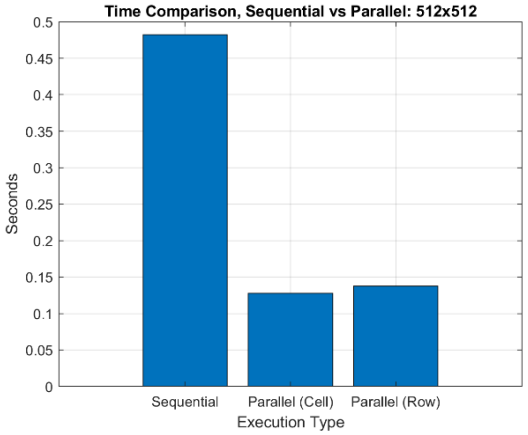
4.2. Comparison between Sequential and OpenMP:

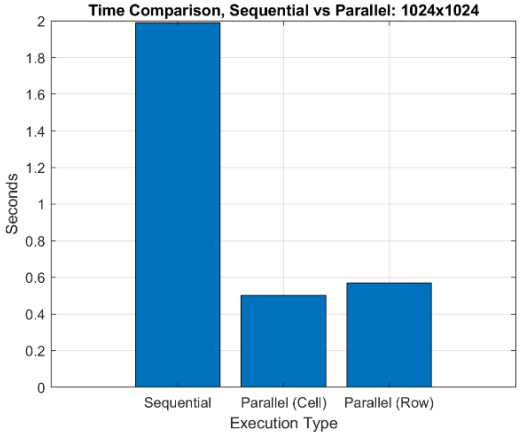
The times for the tests between sequential code, parallel for each cell, and parallel for each row are:

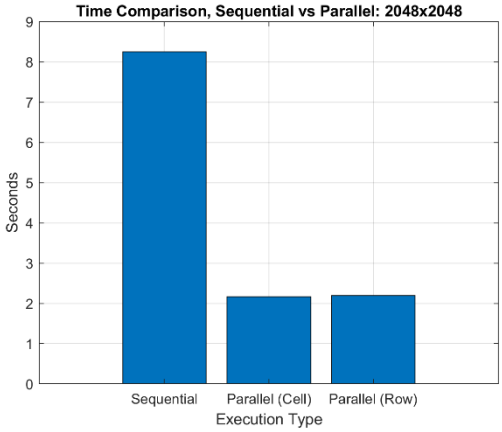
Size	Sequential	OpenMP (cell)	OpenMP (row)
128 × 128	0.03s	0.015s	0.015s
256 × 256	0.122s	0.04s	0.042s
512 × 512	0.482s	0.128s	0.138s
1024 × 1024	1.99s	0.5018s	0.5687s
2048 × 2048	8.25s	2.164s	2.191s









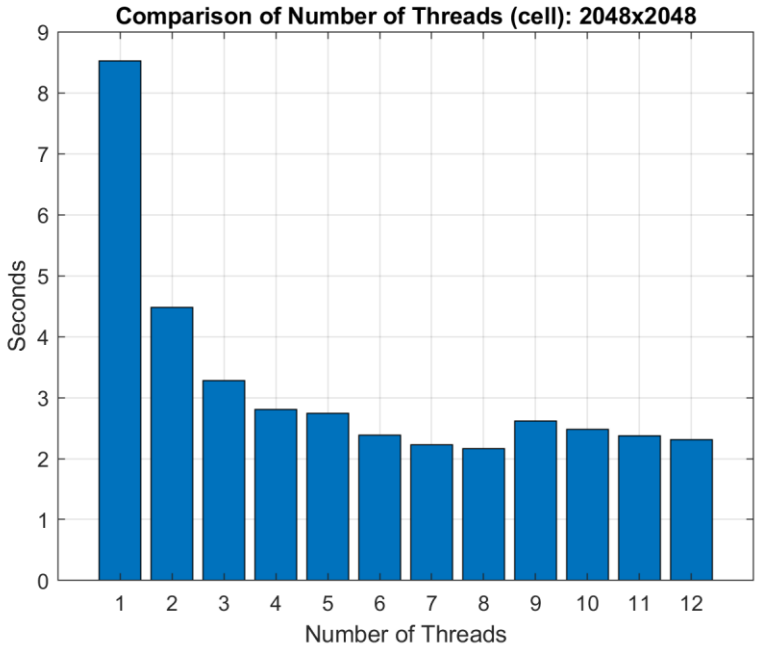


4.3. Comparison between Different Thread Numbers:

The times for tests on different numbers of threads for a size 2048x2048 grid, parallelized for each cell, are:

1-thread	8.5199s
2-threads	4.4775s
3-threads	3.2785s
4-threads	2.807s
5-threads	2.7432s
6-threads	2.3814s
7-threads	2.2299s

8-threads	2.164s
9-threads	2.6191s
10-threads	2.4795s
11-threads	2.3693s
12-threads	2.3107s



5. Discussion:

5.1. Comparison of Parallelization at Different Sections:

The parallel implementations of the canvas generation and initialization did not provide any speed up compared to the sequential implementation. In fact, they were marginally slower than the sequential. This may be due to the overhead introduced by the parallelization, which is not countered by enough reduction of time because the create_canvas() and initialize_canvas() functions only run through the canvas once through.

However, the next_generation implementation was approximately 3.8 times faster than the sequential implementation. This indicates that parallelization in the next_generation() function had the best improvement in the performance of the program. Since in this test the times of the next_generation() row parallelization and cell parallelization were close, the comparison between them was expanded to the next experiment between sequential and OpenMP implementations.

5.2. Comparison between Sequential and OpenMP:

Observation of the statistics of this test reveals that the sequential times were approximately 2 to 4 times faster than the implementations using OpenMP, meaning that parallelizing using OpenMP enhances the program's performance by 2 to 4 times. It can be seen that as the workload of program grows, the improvement on the execution by OpenMP is more pronounced. Parallelization on a size 128 × 128 square grid doubles (0.03 ÷ 0.015) the efficiency of code, while on a size 2048 × 2048 square grid the program is 3.81 times as efficient as its sequential execution after using OpenMP.

When it comes to comparing the times of parallelizing at the grid cell level and row level, the times at the cell level are always slightly faster than parallelizing at the row level. In OpenMP, nested loops

are turned into one big loop if the collapse directive is used. Based on Open MP's effective loop schedule, the big loop is divided into several chunks, and these chunks then are separated to threads. Threads with better performance are allocated with more tasks, and vice versa, in order to get all threads to finish their work at the same time. Because the collapse directive separates the task into the smaller granularity (by cell), the waiting time scale when threads get to the barrier would be on the order of the execution time of a cell instead of on the order of the execution time of a row.
To be more precise, referring back to the statistics above, as the nested for loops are divided to 8 threads in the lab computers, when it comes to the end of parallelization there must be 7 threads waiting for the last thread's execution because of OpenMP's barrier. When the grid size is 128 × 128, at the row level, the 7 threads can be waiting for the last thread to finish executing dozens of cells (to finish the row); while in 2048 × 2048 grid, the 7 threads can be waiting for the last thread to finish executing hundreds of cells. If the collapse directive is used, then no matter how big the size of the grid is, the waiting interval would be always the execution time of few cells. This may explain why the parallelization at the grid cell level is slightly faster than parallelizing at the row level.

5.3. Comparison between Different Thread Numbers:

In this experiment it is seen that the performance increases as the number of threads increases (until 8 threads). 1 thread is the slowest, at 8.5199s, which is comparable to but slightly slower than the sequential time, at 8.25s. As the number of threads increases, the time taken decreases, because the load is split between the threads working in parallel. However, the performance does not increase linearly (for example, 3 threads took 8.5199 ÷ 3.2785 = 2.6 times faster than one thread, slower than 3 times faster as might be expected).

Furthermore, when the number of threads set exceeds 8, the performance drops suddenly. The computers that were tested on have 4 physical cores and 8 logical cores (due to hyperthreading). When the number of threads set is greater than the number of logical cores, there are not enough logical cores to run all the threads simultaneously. Therefore, some threads need to be switched between each other for execution (context switching), introducing extra work for the CPU and slowing the program down. Thus when using 9 or more threads, the performance is seen to have dropped.

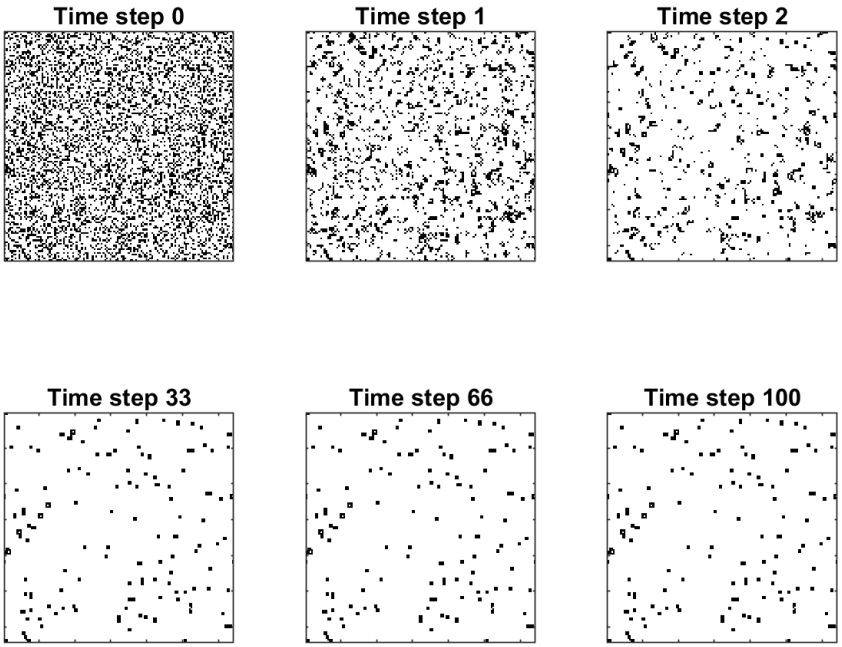
5.4. Conclusion

In conclusion, the fastest way to parallelize the Game of Life explored in this assignment is on the next generations calculations, with 8 threads (the largest number of logical cores in the computer).

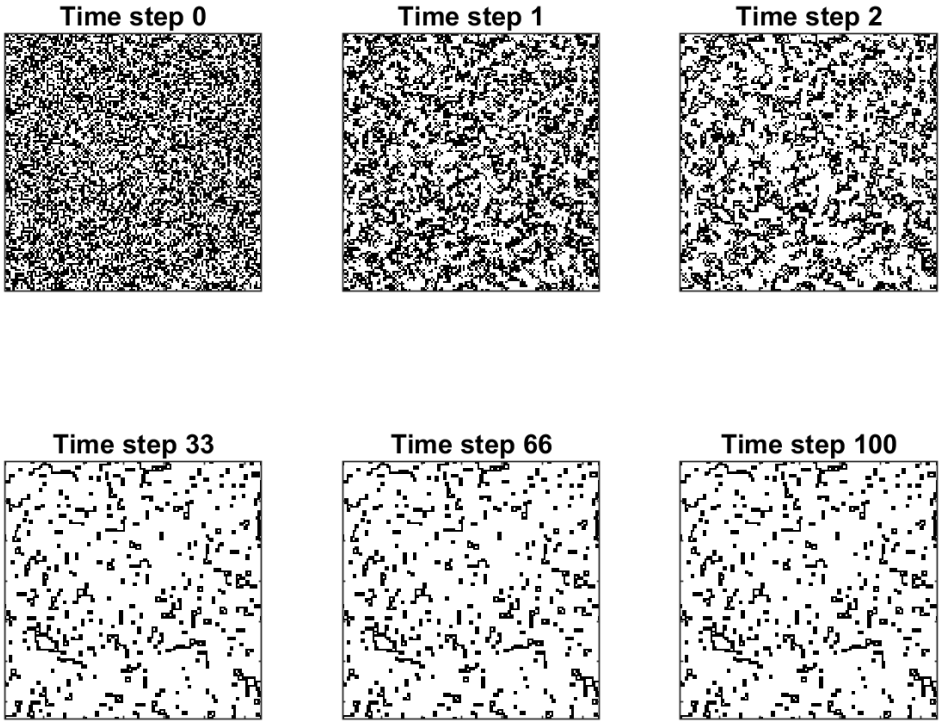
6. Plots of Game States:

For each size of the grid, three different initialization patterns are given by three different probability values (0.3, 0.5, and 0.7) as shown:

128 × 128, P = 0.3

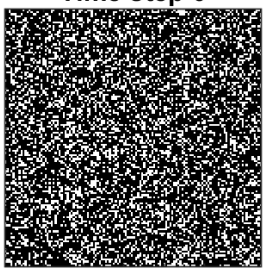


128 × 128, P = 0.5

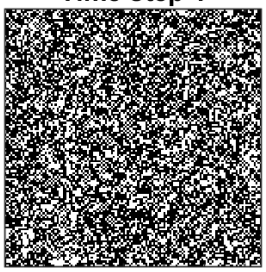


128 × 128, P = 0.7

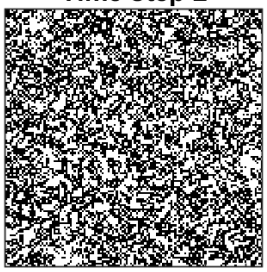
Time step 0



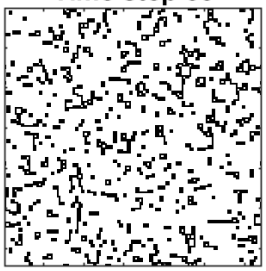
Time step 1



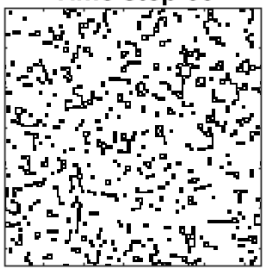
Time step 2



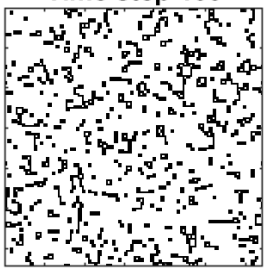
Time step 33



Time step 66

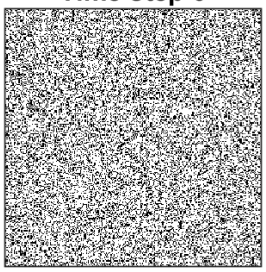


Time step 100

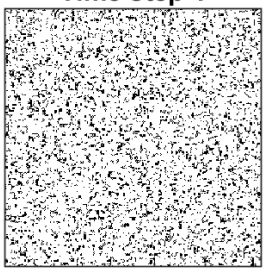


256 × 256, P = 0.3

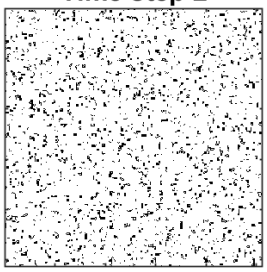
Time step 0



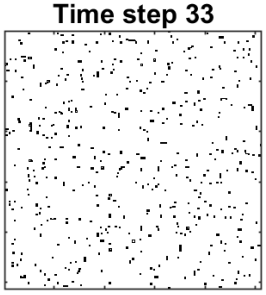
Time step 1



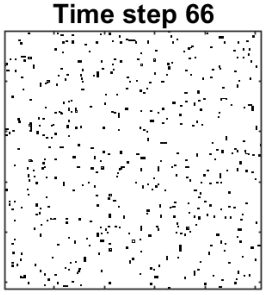
Time step 2



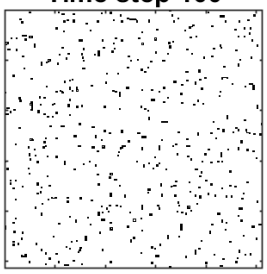
Time step 33



Time step 66

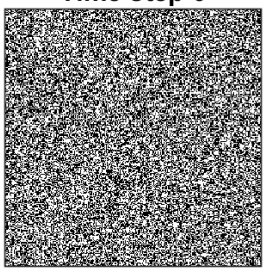


Time step 100

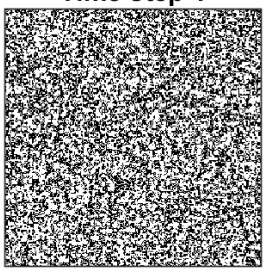


256 × 256, P = 0.5

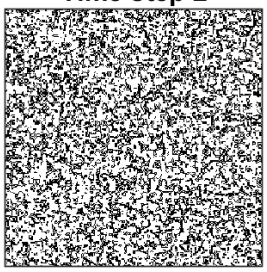
Time step 0



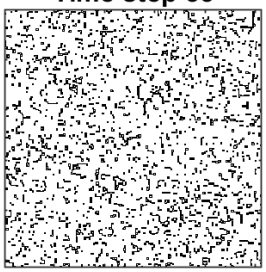
Time step 1



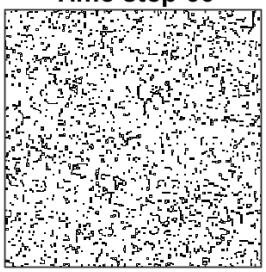
Time step 2



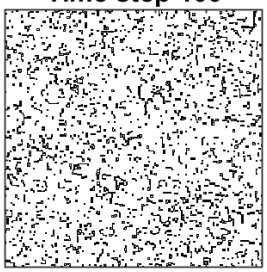
Time step 33



Time step 66

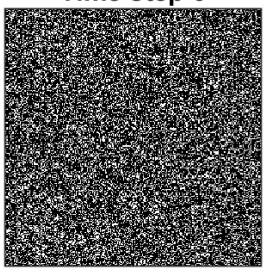


Time step 100

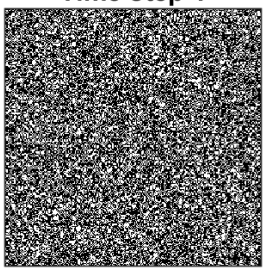


256 × 256, P = 0.7

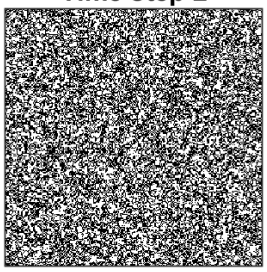
Time step 0



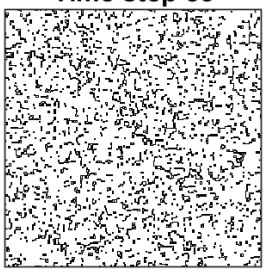
Time step 1



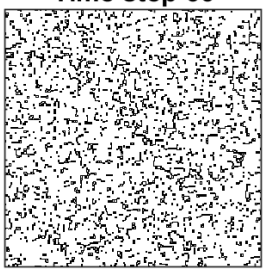
Time step 2



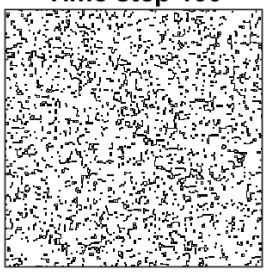
Time step 33



Time step 66

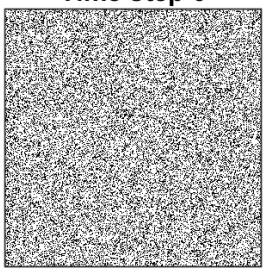


Time step 100

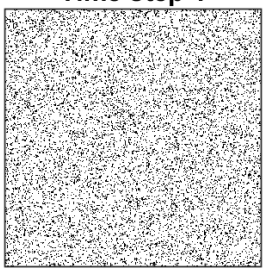


512 × 512, P = 0.3

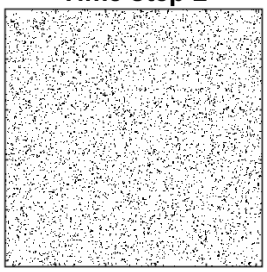
Time step 0



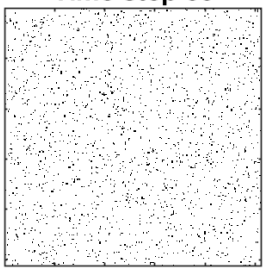
Time step 1



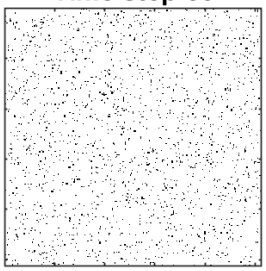
Time step 2



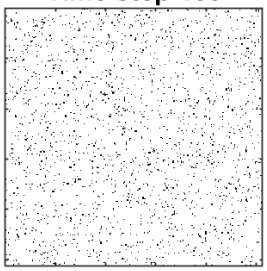
Time step 33



Time step 66

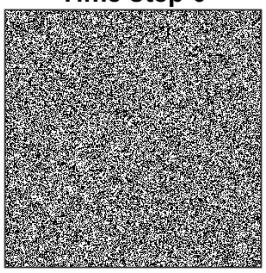


Time step 100

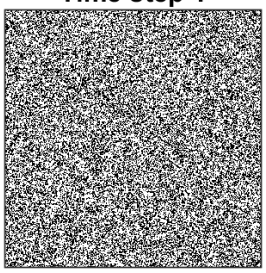


512 × 512, P = 0.5

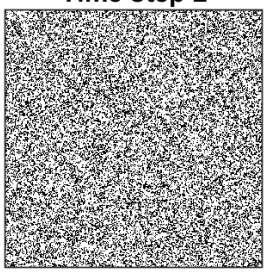
Time step 0



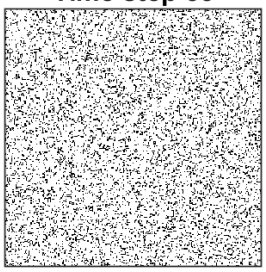
Time step 1



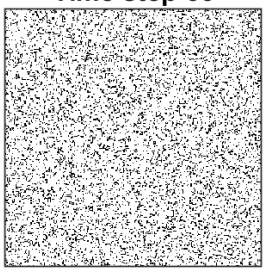
Time step 2



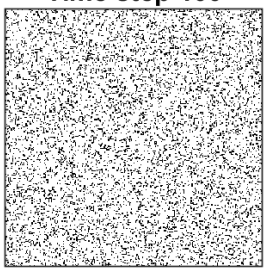
Time step 33



Time step 66

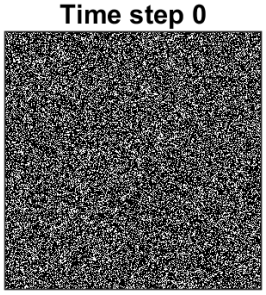


Time step 100

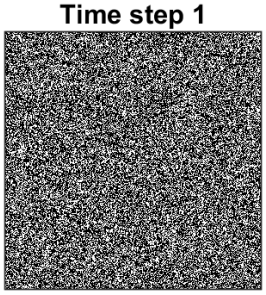


512 × 512, P = 0.7

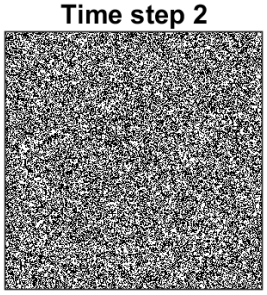
Time step 0



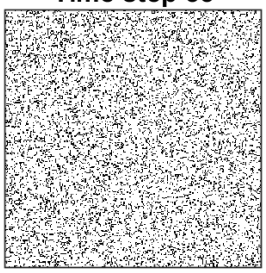
Time step 1



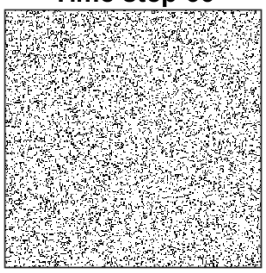
Time step 2



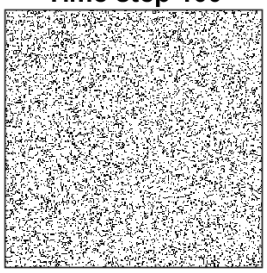
Time step 33



Time step 66

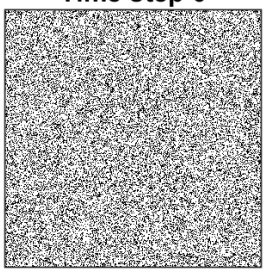


Time step 100

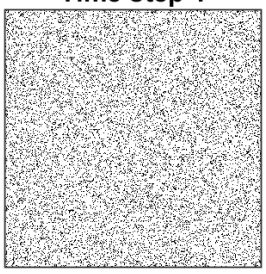


1024 × 1024, P = 0.3

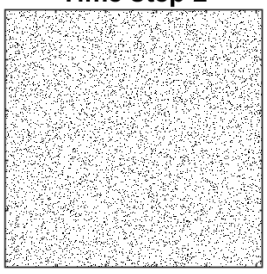
Time step 0



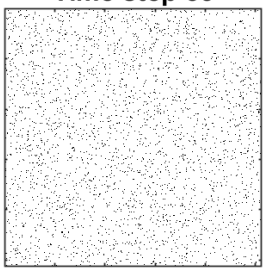
Time step 1



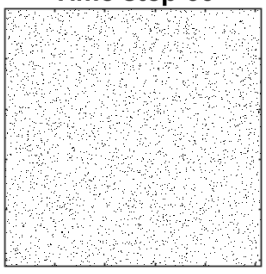
Time step 2



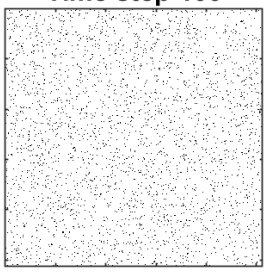
Time step 33



Time step 66

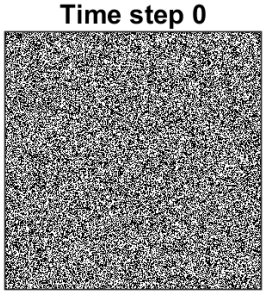


Time step 100

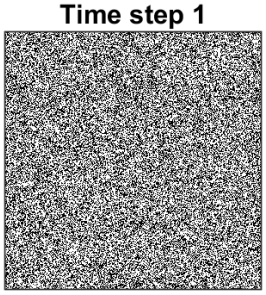


1024 × 1024, P = 0.5

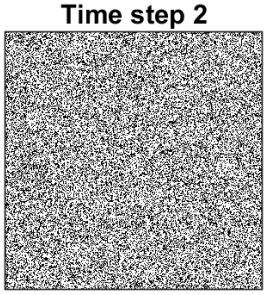
Time step 0



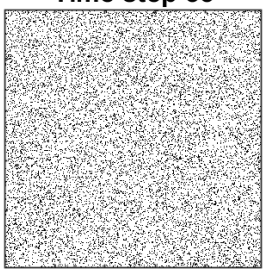
Time step 1



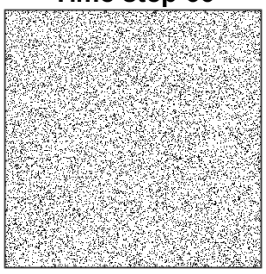
Time step 2



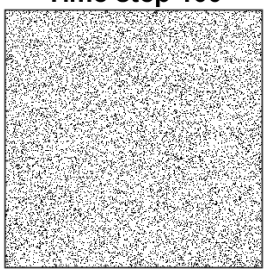
Time step 33



Time step 66

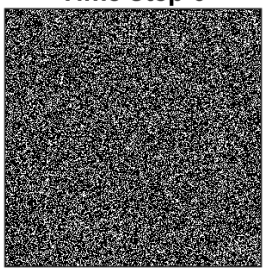


Time step 100

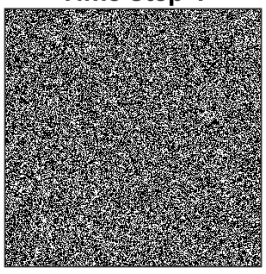


1024 × 1024, P = 0.7

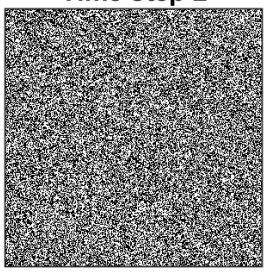
Time step 0



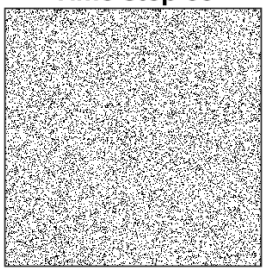
Time step 1



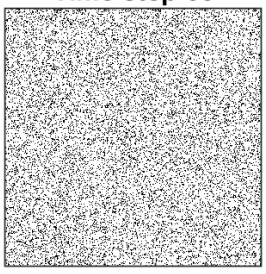
Time step 2



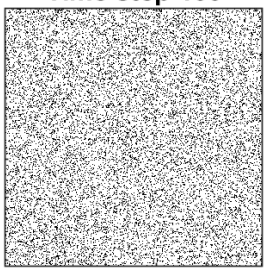
Time step 33



Time step 66

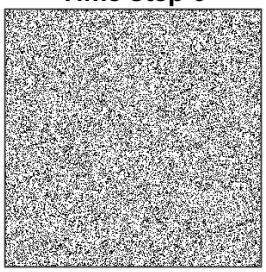


Time step 100

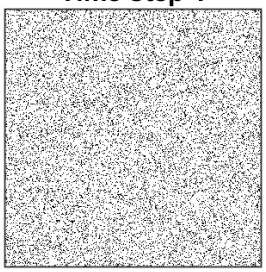


2048 × 2048, P = 0.3

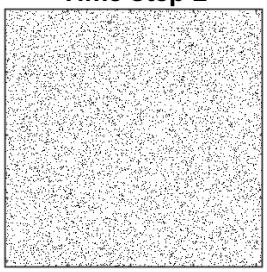
Time step 0



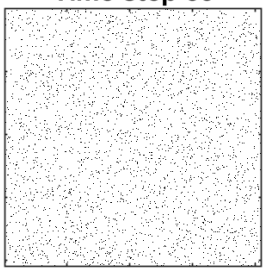
Time step 1



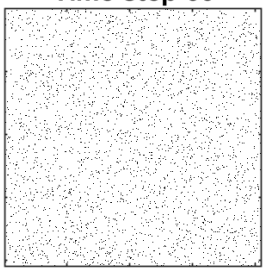
Time step 2



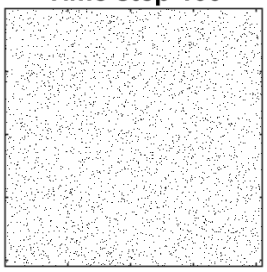
Time step 33



Time step 66

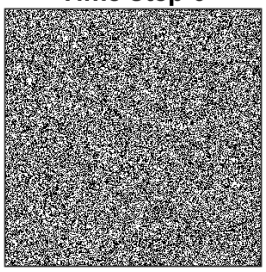


Time step 100

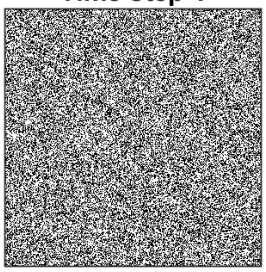


2048 × 2048, P = 0.5

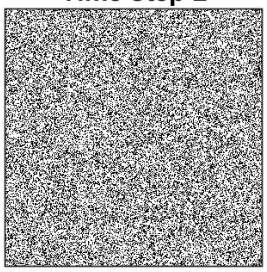
Time step 0



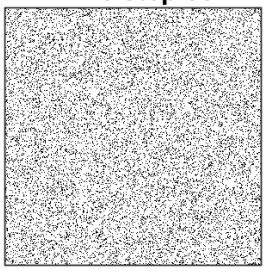
Time step 1



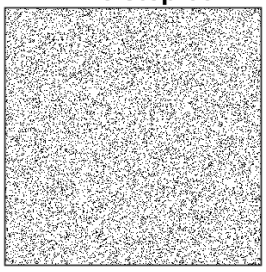
Time step 2



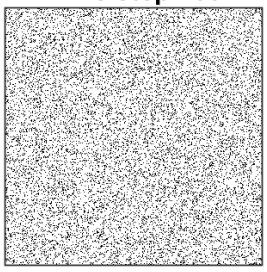
Time step 33



Time step 66

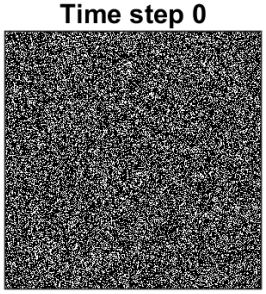


Time step 100

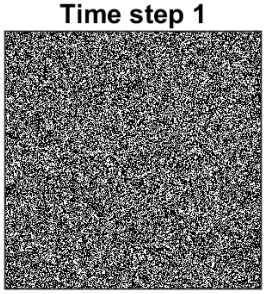


2048 × 2048, P = 0.7

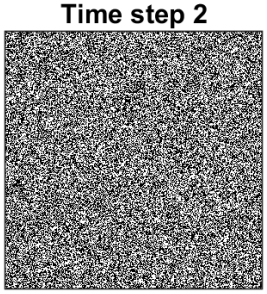
Time step 0



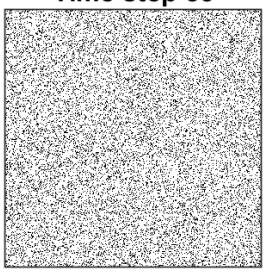
Time step 1



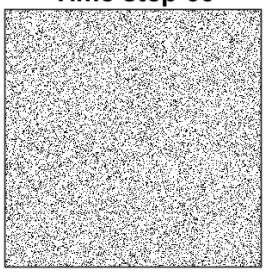
Time step 2



Time step 33



Time step 66



Time step 100