Program Report

(Github Link: https://github.com/ddaib1)

Problem Statement:

We are required to use C Programming to simulate John Horton Conway's "Game of Life". It is a board game with a two-dimensional array of cells. Each cell can be assigned the status of 'alive' or 'dead'. This status is determined by the neighboring cells. The game begins with an initial set up of cells on the array and each update on the array is considered the next generation. With each generation – the status of the cells changes and the game is considered to be over when there is no longer any difference between one generation and the next.

The following rules are taken into consideration when updating the status of a cell:

1. For an 'alive' cell, the following may happen:
    a. If there are one or no 'alive' neighbors, it will die.
    b. If there are four or more 'alive' neighbors, it will die.
    c. If there are two or three 'alive' neighbors, it will survive.

2. For a 'dead' cell, if there are exactly three 'alive' neighbors, it will become 'alive'.

After producing the code for the above problem in serial, we must achieve an optimized algorithm in parallel by using MPI.

Program Design:

We will be utilizing various intricate operations on two-dimensional arrays in C to achieve the simulation of the Game of Life. Before proceeding with our main function – we will design some functions that perform certain operations on two-dimensional arrays as we will have to call back to these functions multiple times. While creating our array, we use the concept of ghost cells for ease of operations as our array is expected to wrap around at each end. We will also be designing and using some other functions that do not operate on the array but are essential to the program.

Another concept that we will utilize is that of ghost cells. Ghost cells are additional cells extended out from each end element of the array where we can copy the corresponding boundary elements to each end of the total array. This helps us in calculating neighboring alive cells when generating each iteration of the matrix. To ensure we have space for the ghost cells – we allocate an array of (N+2)x(N+2) dimensions when the user input for array dimension is NxN. When filling the values of the matrix, we start from (1,1) to (N+1,N+1) instead to ensure a wrap of empty cells to fill with our ghost cells.

We create the following functions for operating on our array(s):

1. *array_alloc* – Used to dynamically allocate memory to initialize our array(s).
2. *array_print* – This simply prints the array.
3. *array_ghostcells* – The function handles the exchange of ghost cells between neighboring processes using non-blocking point-to-point communication. It sends and receives data for the upper and lower rows as well as the left and right columns.
4. *array_copy* – Used to copy the contents of one matrix into another.
5. *array_aresame* – Used to compare the elements of two matrices to check if there is any difference or not.

We also create the following functions for other essential purposes:

1. *count_living_cells* – Used to find the total number of alive neighboring cells near an individual cell.
2. *gettime* – Used to fetch the current system time and convert it into seconds and return the value. This will be used to find the time difference between the start and end times of the program – thus finding overall execution time.

Finally in our main method – we take user input on the dimensions of our matrix as well as the maximum number of generations we will iterate through. We dynamically allocate memory according to user input and initialize two two-dimensional arrays using *array_alloc* that we will refer to A and B for ease. A is filled with elements that are randomly generated to fill it with 1s and 0s where 1 represents an 'alive' cell and 0 represents a 'dead' cell. The wrapping elements are kept empty and we call on *array_ghostcells* to fill them up with the ghost cells.

Before we begin the core part of the game, we save the starting time. We run a loop for the amount of maximum generations specified by the user. For each iteration – we basically run *count_living_cells* for every single cell in matrix A. This will tell us the status of each cell for the next generation – and we store this data in the matrix B.

We check whether or not there is any difference between A and B using *array_aresame* as this will tell us whether there is any difference between two generations or not. Should there be no difference, we break out of the loop as that indicates the end of the game.

Otherwise, we call *array_copy* to copy the content of B into A and run the same loop again – using the cells of A to generate the next iteration in B and so on it continues.

Finally, once either the generations stop changing or the maximum number of generations is reached – we save the end time of the system and print the difference between the start and end time to get the total execution time of the program.

Having completed the basic code, we will then be achieving parallelization by using MPI. The generation of the cells will be in parallel as we will be using send and receive methods inside our aforementioned functions to generate the cells for each iteration. The communication between the various processes is essential to efficiently generate in an acceptable time frame.

We have two versions of the code, one that uses blocking processes and one that uses non-blocking processes, and in the case of both we attempt to obtain an efficient data exchange of one dimensional data where we transfer the data of separate rows and columns of our two dimensional grid.

We use a request array to process all the communication requests and to ensure that all the communication is complete synchronously - we will wait for all of them to finish using a 'Waitall' method. This would be the equivalent of placing a barrier at the end of an OpenMP operation that separates into multiple worker threads.

Test Plans:

To test the correctness of our code after implementing parallelization, we will run the output for some small generations. We will run the generations with the exact same specifications and the initial generation is done with the same seed randomizer in both the serial and parallel version of the code. Then we can compare the output of the two and check each cell to see if the same generations are produced. If the generations of both serial and parallel are the same – then it confirms the correctness of our code.

To see the efficiency of the code run in a parallel system – we will use the specifications of a matrix of 5000x5000 cells with a maximum of 5000 generations. Then we will run the code multiple times with the following as input for number of threads: 2, 4, 8, 10, 16, and 20. We note the time taken for each of those iterations and calculate speedup and efficiency. We plot the same for a visual representation to get an idea in the disparity of the run times for serial and parallel codes for various threads. We have a record of the code run times and efficiency from the serial and OpenMP parallel implementations of the code, and we will plot and compare those values to the observations we get from the MPI parallel implementation.

Test Cases:

Test Cases:

For the Game of Life in C program running in serial, we note the following run time:

Test Case #	Problem Size	Max Generations	Time Taken
1	5000x5000	5000	4787.989459 s

And in a similar vein, we run it in parallel using OpenMP to note the following run times for the number of threads: (The specifications of the problem size and generations is the same as serial)

Test Case #	No. of Threads	Time Taken	Speedup	Efficiency
1	2	2886.791473 s	1.658	82.90%
2	4	2015.071365 s	2.376	59.40%
3	8	1900.685905 s	2.519	31.48%
4	10	1897.445621 s	2.523	25.23%
5	16	1894.611733 s	2.527	15.79%
6	20	1891.771245 s	2.531	12.65%

Speedup = (Time taken in Serial) / (Time taken in Parallel)

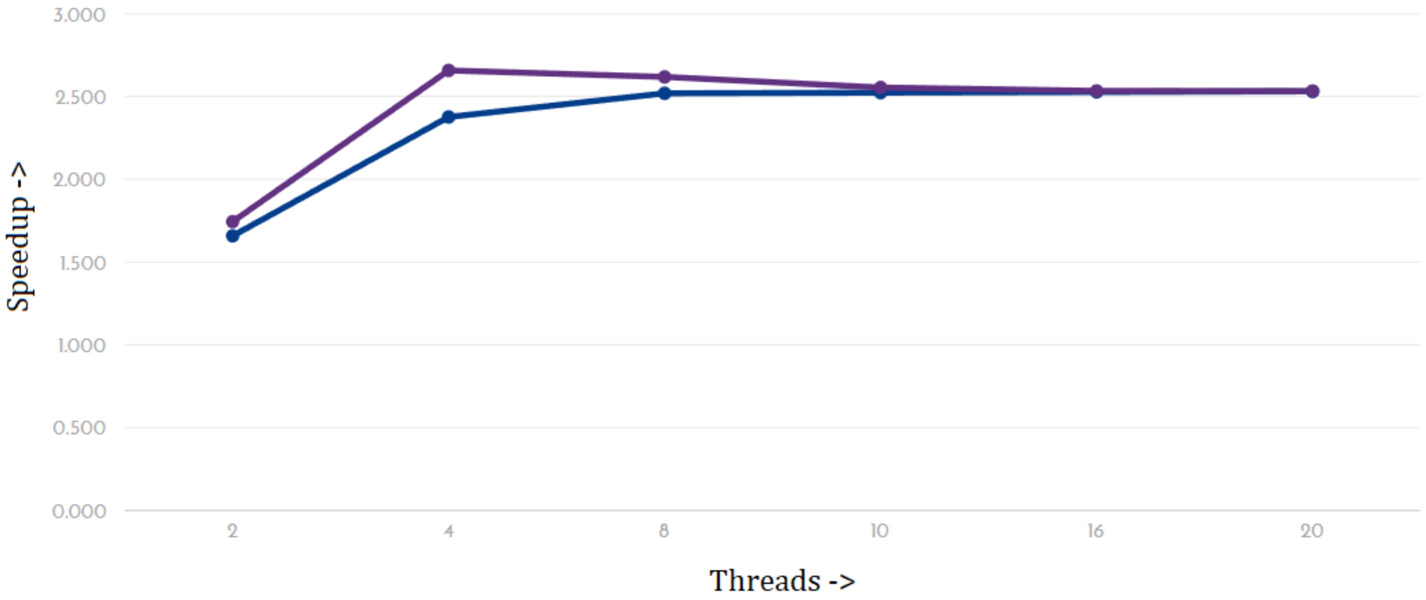
Efficiency = (Time taken in Serial) / [(No. of Cores) * (Time taken in Parallel)] = Speedup / (No. of Cores)

Finally, for MPI with blocking message passing, we can note the following run times:
(Once again the specifications remain unchanged)

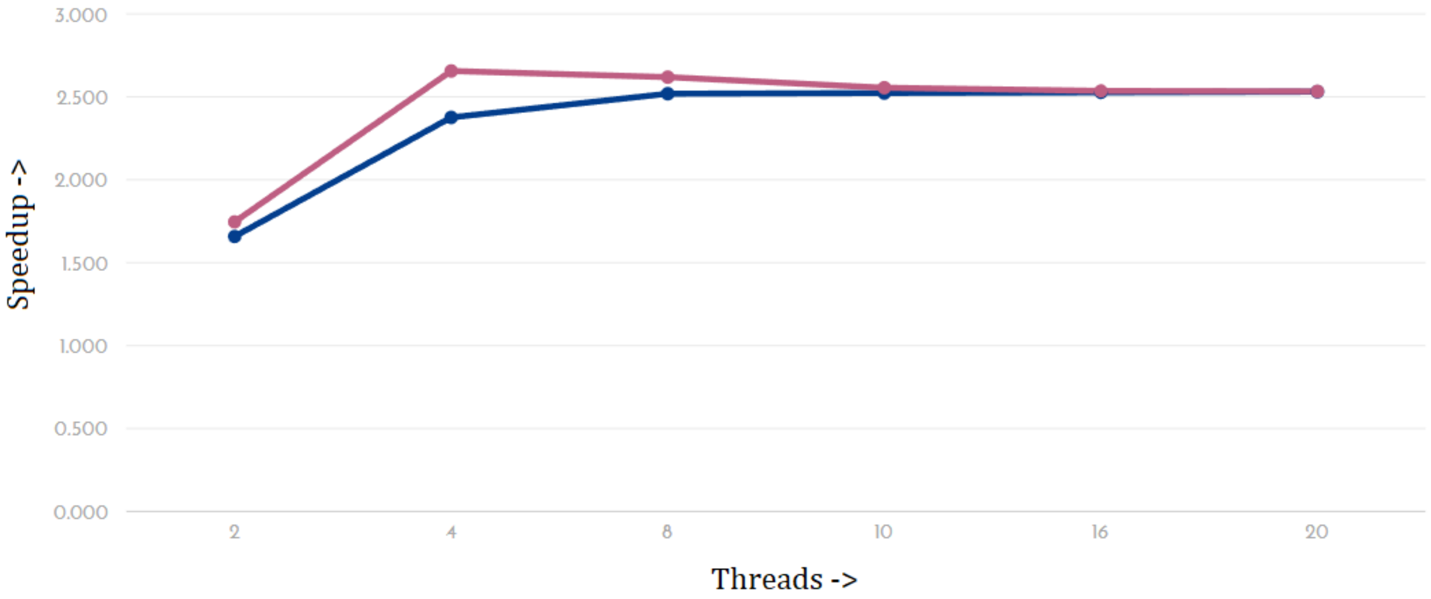
Test Case #	No. of Threads	Time Taken	Speedup	Efficiency
1	2	2746.141589 s	1.744	87.20%
2	4	1802.221873 s	2.657	66.43%
3	8	1828.492876 s	2.618	32.73%
4	10	1874.918724 s	2.554	25.54%
5	16	1889.245261 s	2.534	15.84%
6	20	1890.712154 s	2.532	12.66%

Finally, for MPI with non-blocking message passing, we can note the following run times:
(Once again the specifications remain unchanged)

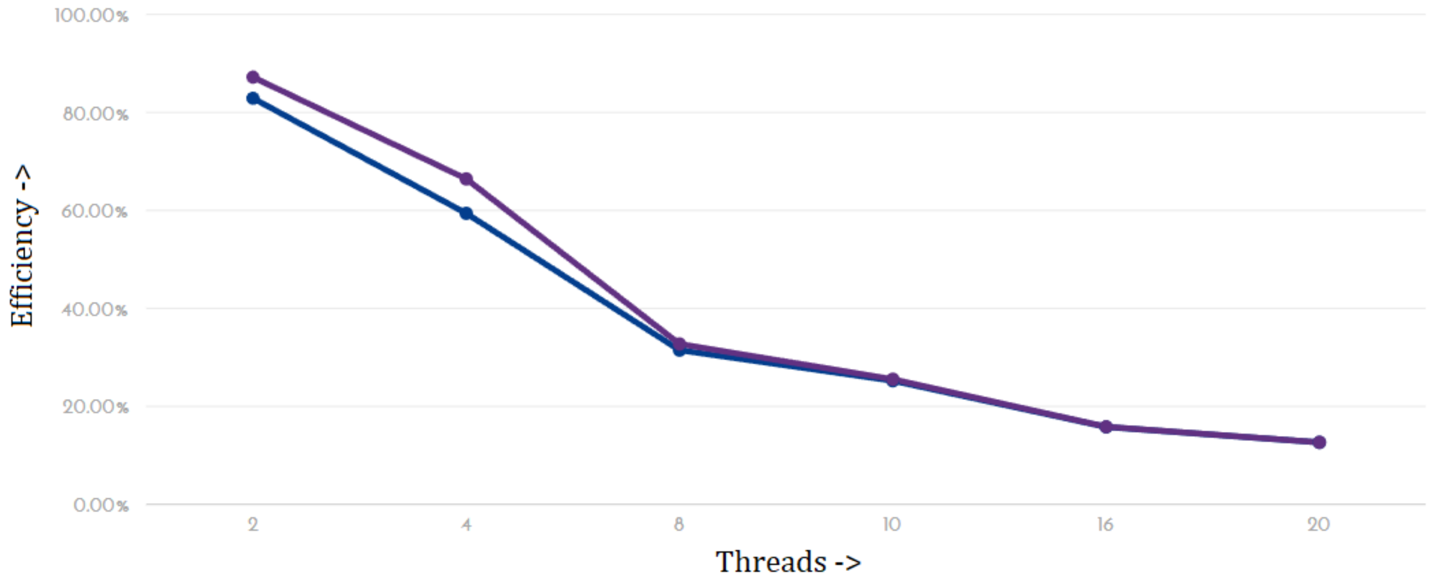
Test Case #	No. of Threads	Time Taken	Speedup	Efficiency
1	2	2742.325872 s	1.746	87.30%
2	4	1804.238527 s	2.656	66.41%
3	8	1827.912875 s	2.619	32.74%
4	10	1873.623721 s	2.555	25.55%
5	16	1887.989447 s	2.536	15.85%
6	20	1889.977264 s	2.533	12.67%



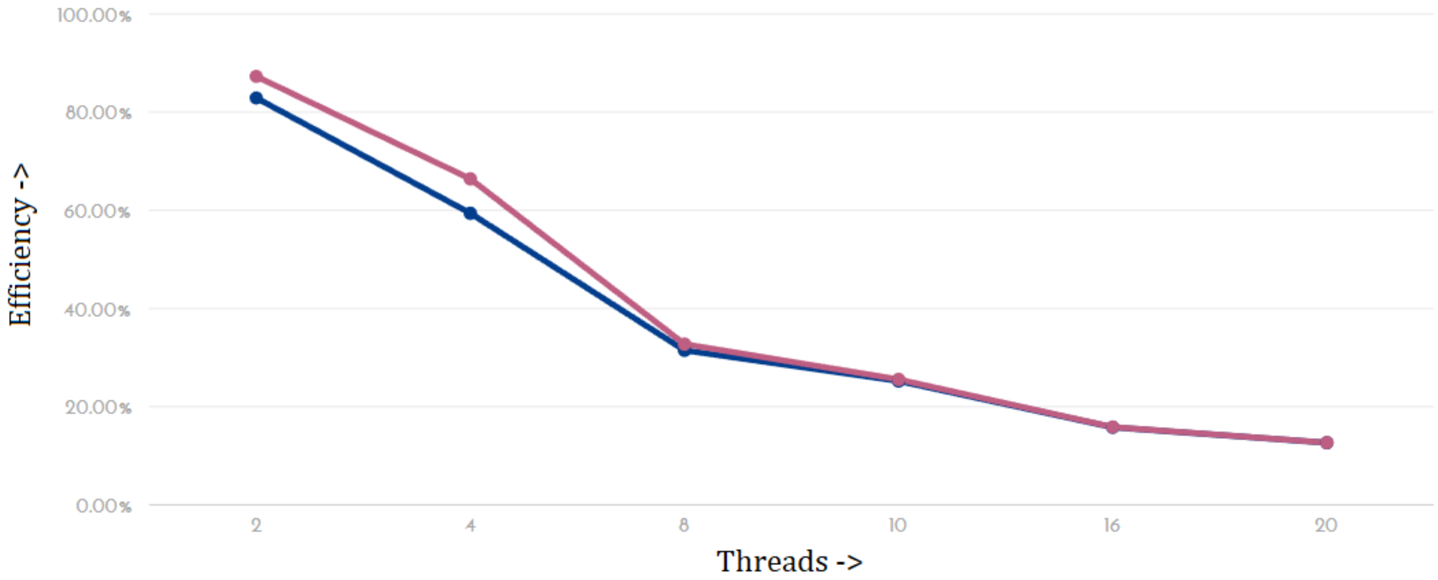
Graph of speedup vs. number of threads. Where the blue nodes indicate the speedup obtained using OpenMP and the purple nodes indicate the speedup obtained using blocking MPI.



Graph of speedup vs. number of threads. Where the blue nodes indicate the speedup obtained using OpenMP and the pink nodes indicate the speedup obtained using non-blocking MPI.



Graph of efficiency vs. number of threads. Where the blue nodes indicate the speedup obtained using OpenMP and the purple nodes indicate the speedup obtained using blocking MPI.



Graph of efficiency vs. number of threads. Where the blue nodes indicate the speedup obtained using OpenMP and the pink nodes indicate the speedup obtained using non-blocking MPI.

Analysis and Conclusions:

So after noting down the above observations – we can confirm the correctness of our program as proven by checking the output of both serial and the variouus parallel versions and finding them all to be the exact same.

Moving on to the execution time, we take the a base specification of 5000x5000 matrix running for 5000 generations at max. With that we run the program for the above noted number of threads and notice a distinct pattern. Right off, we can see that for 1 thread (serial), the run time is 4787.989459 seconds. If we increase the number of threads to 2, we can see an instant decrease in run time to a fair bit more than half the time to

2746.141589 seconds. This is seen to be highly efficient with only twice the usual threads are dedicated to the task. This is noticeably also less than the time taken in OpenMP which was 2886.791473 seconds.

Now moving on to twice even that – to 4 threads, the time reduction is 1802.221873 seconds – where the speedup is not as proportionally good as the one obtained from 2 threads with a notable decrease in efficiency, however it is still a significant decrease. Once again, there is even more of a reduction from the time taken in OpenMP of 2015.071365 seconds. Moving on to 8 threads, the decrease in time is already much less significant thus giving us a far less efficient code. This gives us the trend going forward – as with 10 threads, 16 threads and 20 threads; we can see diminishing returns where the efficiency only continues to decrease. So we can see our speedup and efficiency plateaus after 8 threads. Thus an ideal number of threads would be 2 or possibly 4.

Now, for all of the times noted for blocking MPI message passing, we will note that the non-blocking MPI message passing is slightly less but not significantly so. This is because even for non-blocking MPI message passing, we use the 'Waitall' function to wait for all the messages to finish passing, therefore the wait time would roughly end up being the same for both blocking and non-blocking. This wait time is unavoidable as we have to ensure correctness of the output even in synchronous programming, therefore not much of a significant change was noticed in the speedup and efficiency values.