Project Report

on

Parallel
Conway's Game of life

Tuhin Khare (140905056)
Harshil Gupta (140905314)

Abstract

This project deals with

- Conway's Game of Life as an Example of cellular automaton.
- Parallel Implementation of Conway's Game of Life.

Upon completion of this project, there should be a clear understanding of working of cellular automaton. How different types of shapes work in cellular automation for example glider, spaceship etc. And, on what rule sets does Conway's Game of life works.

Alongside, a deeper insight is gained on the parallelisation of game of life, and on how a given number of processes can simulatneously work towards generating a simulation of game of life, also reducing the time taken by the automata to reach stability along with increasing the speedup given by the Amdahl's Law.

Motivation

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time. Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit on one computer. Only one instruction may execute at a time. Once that instruction is finished, the next one is executed.

Parallel computing, on the other hand, uses multiple processing elements simultaneously to solve a problem. In parallel processing, instead of a single program executing tasks in a sequence, the program is split among multiple "execution flows" executing tasks in parallel, i.e. at the same time.

These execution flows can be either shared memory models (threads) or distributed memory models which communicate through send and receive messages. In this project we will follow a distributed parallel model in order to achieve the purpose i.e a particular number of processes will be chosen and will be given a subtask. The major factors for choosing parallelism are.

- **“Speedup”** is the idea that a program will run faster if it is parallelised as opposed to executed serially. The advantage of speedup is that it allows a problem to be modeled faster. If multiple execution flows are able to work at the same time, the work will be finished in less time than it would take a single execution flow. Speedup is an enticing advantage.
- **“Accuracy”** is the idea of forming a better solution to a problem. If more processes are assigned to a task, they can spend more time doing error checks or other forms of diagnostics to ensure that the final result is a better approximation of the problem that is being modeled. In order to make a program more accurate, speedup may need to be sacrificed.
- **“Scaling”** is perhaps the most promising of the three motivations. Scaling is the concept that more parallel processors can be used to model a bigger problem in the same amount of time it would take fewer parallel processors to model a smaller problem. A common analogy to this is that one person in one boat in one hour can catch a lot fewer fish than ten people in ten boats in one hour.

Objectives

Statement - Implement a proof of concept program which results in a basic simulation of conway’s cellular automata, using a distributed parallel model, using CUDA and OpenMPI.

Measures - The indicators of success are when a random grid of cells consisting of cells in states = {ON, OFF} is given, it applies the given set of rules for the automata successively and reaches a stable state itselff upon which when the rules are applied there is no generation of further states.

Performance Measures - The performance is compared using Amdahl’s Law of speedup of parallel algorithms as compared to the sequential algorithms which is given by

$$\textbf{Speedup} = 1/(1-P + (P/N))$$

where

P = is the proportion of the program that can be made in parallel and

N = the number of processes

Introduction

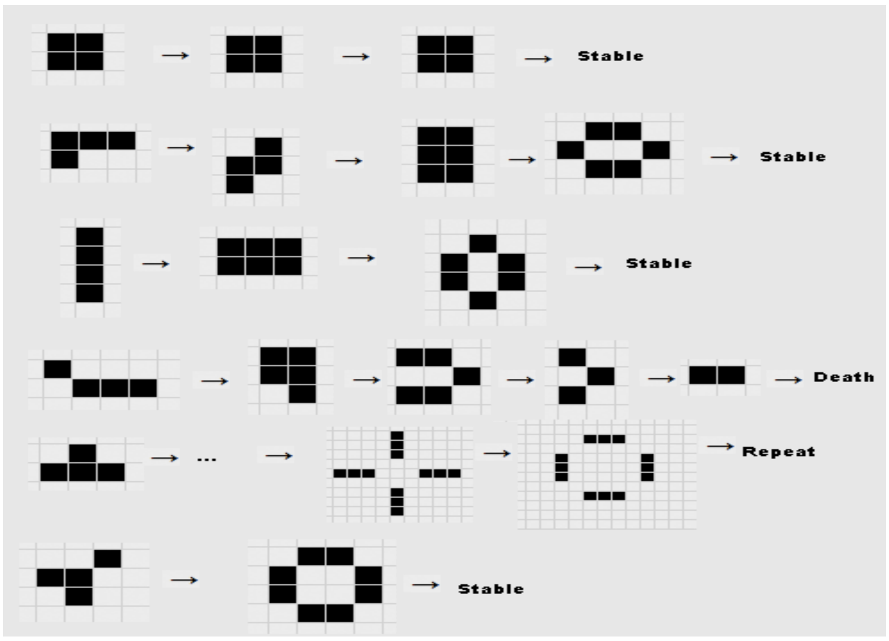
The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970

The game of life is just one example of a **cellular automaton**, which is any system in which rules are applied to cells and their neighbours in a regular grid.

Each cell can be either alive or dead. The status of each cell changes each turn of the game (also called a generation) depending on the statuses of that cell's 8 neighbours. Neighbours of a cell are cells that touch that cell, either horizontal, vertical, or diagonal from that cell.
The initial pattern is the first generation. The second generation evolves from applying the rules simultaneously to every cell on the game board, i.e. births and deaths happen simultaneously. Afterwards, the rules are iteratively applied to create future generations. For each generation of the game, a cell's status in the next generation is determined by a set of rules. These simple rules are as follows:
- If the cell is alive, then it stays alive if it has either 2 or 3 live neighbours.
- If the cell is dead, then it springs to life only in the case that it has 3 live neighbours.

There are, of course, as many variations to these rules as there are different combinations of numbers to use for determining when cells live or die. Conway tried many of these different variants before settling on these specific rules. Some of these variations cause the populations to quickly die out, and others expand without limit to fill up the entire universe, or some large portion thereof. The rules above are very close to the boundary between these two regions of rules, and knowing what we know about other chaotic systems, you might expect to find the most complex and interesting patterns at this boundary, where the opposing forces of runaway expansion and death carefully balance each other. Conway carefully examined various rule combinations according to the following three criteria:

- There should be no initial pattern for which there is a simple proof that the population can grow without limit.
- There should be initial patterns that apparently do grow without limit.
- There should be simple initial patterns that grow and change for a considerable period before coming to an end in the following possible ways:
    1. Fading away completely (from overcrowding or becoming too sparse)
    2. Settling into a stable configuration that remains unchanged thereafter, or entering an oscillating phase in which they repeat an endless cycle of two or more periods.



Literature Review

The Cellular automaton is an important tool in science that can be used to model a variety of natural phenomena.
Cellular automaton can be used:

1. To simulate Brain Tumour growth by generating 3-D image of brain and advancing cellular growth over time.
2. In ecology, cellular automaton is used to model interaction of species competing for resources.

Cellular automaton is also used to design public key cryptosystem. A public-key cryptosystem is a cryptographic protocol that relies on two keys – an enciphering key E, which is made public, and a deciphering key D, which is kept private.

Some other examples where cellular automaton is used are cognitive science, hydrodynamics, agriculture, dermatology etc.

Methodology

This project is implemented using Compute Unified Device Architecture (CUDA) and Message passing Interface(MPI).

Message Passing Interface (MPI) Implementation

A grid of a specific dimensions consisting of cells is taken, which is a unidimensional character array, the states of the cells are defined by two characters

- '#' := The cell is alive
- '.' := The cell is dead

In the grid every cell consists of internal cell consists of 8 adjacent neighbours except for the cells at the edge, which consists of either 3 neighbours if it exists at the corners or 5 neighbours if it exists at the edge, to encounter this problem the concept of "ghost rows" are used which are mainly 4 additional arrays which signify 2 rows and 2 columns of each of the four edges of the grid respectively. The rule set is applied on the ghost row when it is paired with the row which is directly opposite to that particular ghost row, this creates an abstract toroidal wrap around the grid such that the basic non terminating grid configurations such as glider do not terminate when the code is executed.

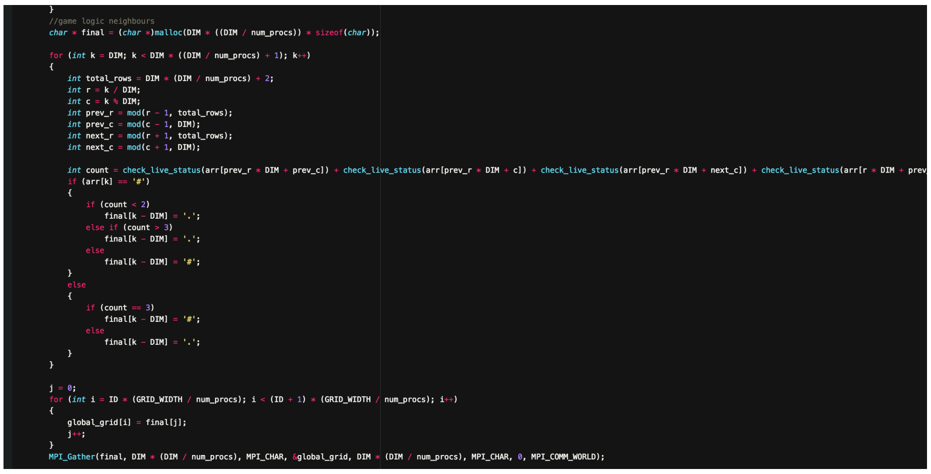
The data parallelism is exploited by assigning groups of rows together to processes, this is in direct proportion to the number of processes given by the user in the input. The number of processes should be such that the total size of the grid is divisible by the number of processes.

((grid_dimension)^2) mod (number of processes) == 0

Since the state of every cell depends on its adjacent neighbours which are governed by successive processes, the communication is done in Synchronous send mode, and the count of each alive cell in the neighbourhood of that cell is calculated along with the grid updation process.

Finally, the updated grid is printed at each step when the rule set is applied for each of the cell.

Code Snippets

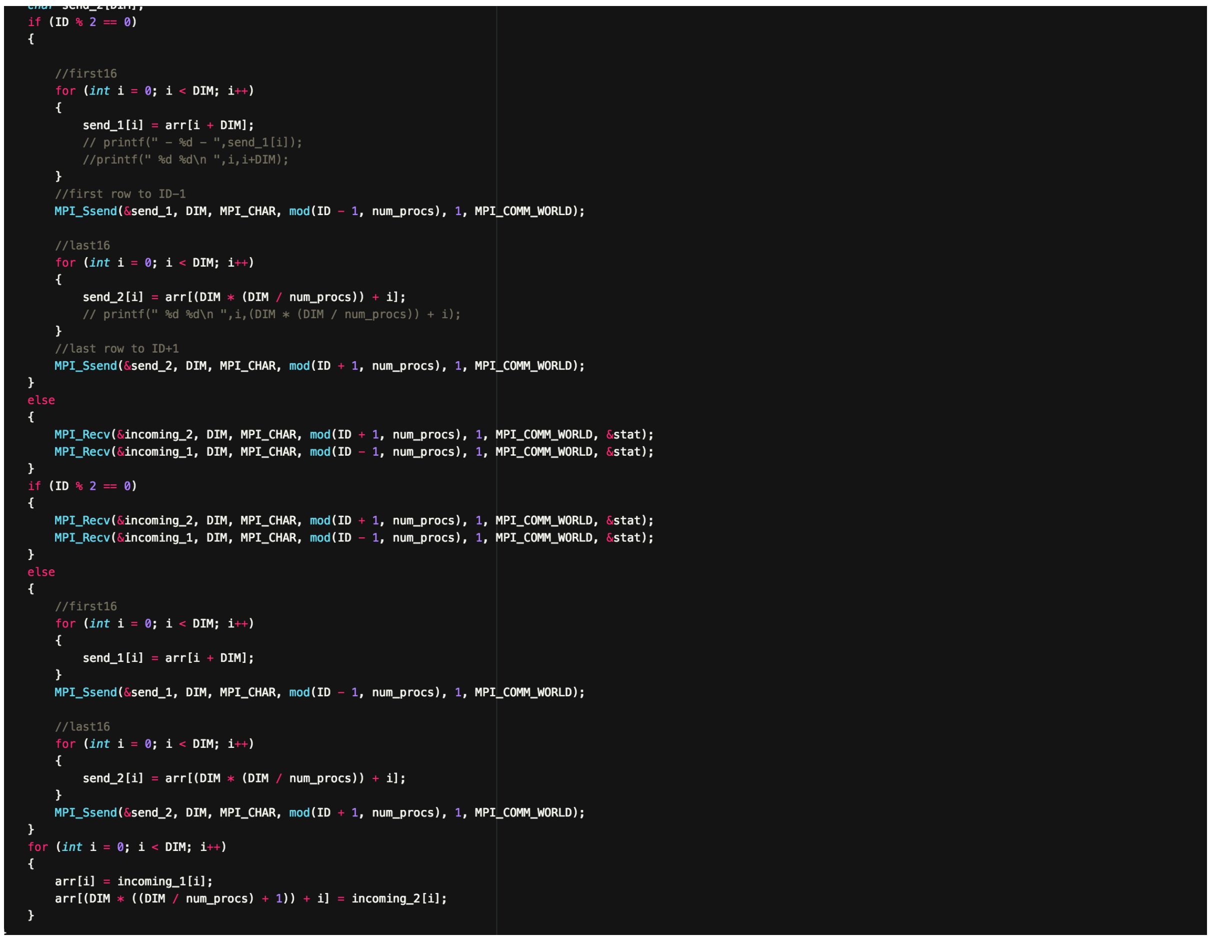


```
    }
    //game logic neighbours
    char * final = (char *)malloc(DIM * ((DIM / num_procs)) * sizeof(char));

    for (int k = DIM; k < DIM * ((DIM / num_procs) + 1); k++)
    {
        int total_rows = DIM * (DIM / num_procs) + 2;
        int r = k / DIM;
        int c = k % DIM;
        int prev_r = mod(r - 1, total_rows);
        int prev_c = mod(c - 1, DIM);
        int next_r = mod(r + 1, total_rows);
        int next_c = mod(c + 1, DIM);

        int count = check_live_status(arr[prev_r * DIM + prev_c]) + check_live_status(arr[prev_r * DIM + c]) + check_live_status(arr[prev_r * DIM + next_c]) + check_live_status(arr[r * DIM + prev
        if (arr[k] == '#')
        {
            if (count < 2)
                final[k - DIM] = '.';
            else if (count > 3)
                final[k - DIM] = '.';
            else
                final[k - DIM] = '#';
        }
        else
        {
            if (count == 3)
                final[k - DIM] = '#';
            else
                final[k - DIM] = '.';
        }
    }

    j = 0;
    for (int i = ID * (GRID_WIDTH / num_procs); i < (ID + 1) * (GRID_WIDTH / num_procs); i++)
    {
        global_grid[i] = final[j];
        j++;
    }
    MPI_Gather(final, DIM * (DIM / num_procs), MPI_CHAR, &global_grid, DIM * (DIM / num_procs), MPI_CHAR, 0, MPI_COMM_WORLD);
```



```
if (ID % 2 == 0)
{

    //first16
    for (int i = 0; i < DIM; i++)
    {
        send_1[i] = arr[i + DIM];
        // printf(" - %d - ",send_1[i]);
        //printf(" %d %d\n ",i,i+DIM);
    }
    //first row to ID-1
    MPI_Ssend(&send_1, DIM, MPI_CHAR, mod(ID - 1, num_procs), 1, MPI_COMM_WORLD);

    //last16
    for (int i = 0; i < DIM; i++)
    {
        send_2[i] = arr[(DIM * (DIM / num_procs)) + i];
        // printf(" %d %d\n ",i,(DIM * (DIM / num_procs)) + i);
    }
    //last row to ID+1
    MPI_Ssend(&send_2, DIM, MPI_CHAR, mod(ID + 1, num_procs), 1, MPI_COMM_WORLD);
}
else
{
    MPI_Recv(&incoming_2, DIM, MPI_CHAR, mod(ID + 1, num_procs), 1, MPI_COMM_WORLD, &stat);
    MPI_Recv(&incoming_1, DIM, MPI_CHAR, mod(ID - 1, num_procs), 1, MPI_COMM_WORLD, &stat);
}
if (ID % 2 == 0)
{
    MPI_Recv(&incoming_2, DIM, MPI_CHAR, mod(ID + 1, num_procs), 1, MPI_COMM_WORLD, &stat);
    MPI_Recv(&incoming_1, DIM, MPI_CHAR, mod(ID - 1, num_procs), 1, MPI_COMM_WORLD, &stat);
}
else
{
    //first16
    for (int i = 0; i < DIM; i++)
    {
        send_1[i] = arr[i + DIM];
    }
    MPI_Ssend(&send_1, DIM, MPI_CHAR, mod(ID - 1, num_procs), 1, MPI_COMM_WORLD);

    //last16
    for (int i = 0; i < DIM; i++)
    {
        send_2[i] = arr[(DIM * (DIM / num_procs)) + i];
    }
    MPI_Ssend(&send_2, DIM, MPI_CHAR, mod(ID + 1, num_procs), 1, MPI_COMM_WORLD);
}
for (int i = 0; i < DIM; i++)
{
    arr[i] = incoming_1[i];
    arr[(DIM * ((DIM / num_procs) + 1)) + i] = incoming_2[i];
}
```

Compute Unified Device Architecture (CUDA) Implementation

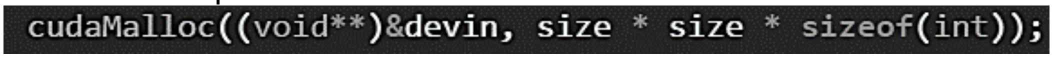
This part gives the methodology for implementing Game of life parallelly in CUDA. The GPU being Nvidia Geforce GTX 960m which has 640 cuda cores.

In Conway's game of life, the input array for demonstration is random. Size of the array is taken from the user. The number of iteration to be performed is kept large because we don't know that when the stability will be achieved.

C++ has a function 'rand ()' which randomly allocated a number and taking the mod of this 'rand()%2' will give us dead(0) or alive(1) cells which are also in random order.

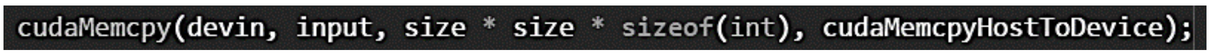
Input: 1-D array of size= N * N

Here size is equal to N.



```
cudaMalloc((void**)&devin, size * size * sizeof(int));
```

So we first use ***cudamalloc ()*** to allocate memory of size equal to the array size. So similarly, space is allocated to the output matrix.



```
cudaMemcpy(devin, input, size * size * sizeof(int), cudaMemcpyHostToDevice);
```

Now ***cudaMemcpy()*** is used to copy the input matrix to the memory of the device. ***cudaMemcpyHostToDevice*** tells that the array should be copied from the host to the device or the memory in the GPU.

Similarly, ***cudaMemcpyDeviceToHost*** tells that the output which was computed in the GPU memory is to be copied from the Device to the host memory.

```
cudaMemcpy(output, devout, size * size * sizeof(int), cudaMemcpyDeviceToHost);
```

Now the number of threads is equal to the number of elements in a row. And number of block is also equal to the number of columns. So conclusively, there are ***N*** threads in a Block and there are ***N*** blocks. Thus, there are ***N * N*** threads in total.

```
cgol<<<no_blocks, no_threads>>>(devtemp, devout, size);
```

For freeing up the memory used up in computation we use ***cudaFree()***

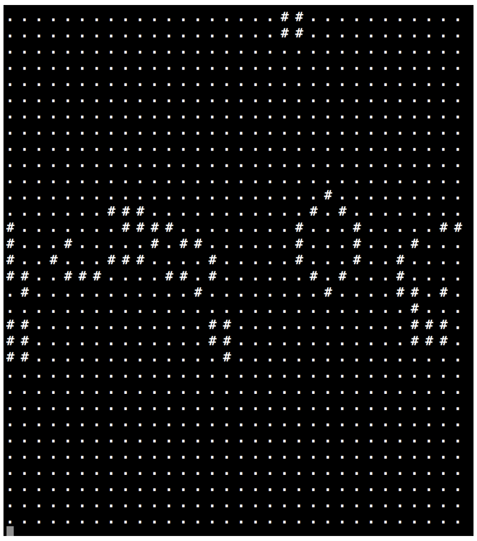
```
cudaFree(devin);
cudaFree(devout);
cudaFree(devtemp);
```

Now in the kernel code, we apply the rule set that Conway suggested. For each cell in the array this ruleset is applied and only then it's state in the next iteration is suggested

Results

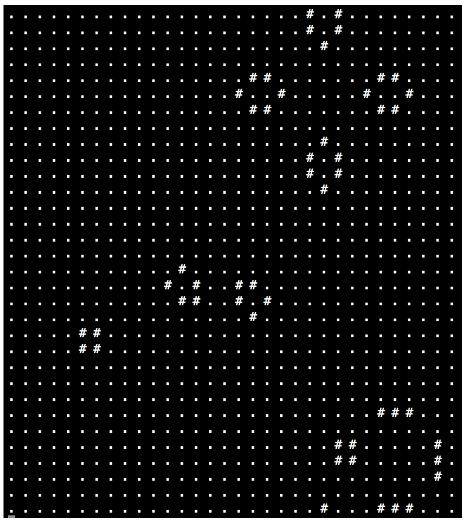
The results is a grid in which the automata ruleset is applied continuously on each and every cell and the grid is updated accordingly at each step until the stable state is achieved where on application of the ruleset the state does not change any further.

Intermediate Stage



```
. . . . . . . . . . . . . . . . . . . # # . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . # # . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . # . . . . . . . . .
. . . . . . . # # # . . . . . . . . . . . # . # . . . . . . . .
# . . . . . . . # # # # . . . . . . . . # . . . # . . . . . # #
# . . . # . . . . . # . # # . . . . . . # . . . # . . . # . . .
# . . # . . . # # # . . . . # . . . . . # . . . # . . # . . . .
# # . . # # # . . . . # # . # . . . . . . # . # . . . # . . . .
. # . . . . . . . . . . . # . . . . . . . . # . . . . # # . # .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . # . . .
# # . . . . . . . . . . . . # # . . . . . . . . . . . . # # # .
# # . . . . . . . . . . . . # # . . . . . . . . . . . . # # # .
# # . . . . . . . . . . . . . # . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

Stable State



```
. . . . . . . . . . . . . . . . . . . . . # . # . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . # . # . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . # . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . # # . . . . . . . # # . . . .
. . . . . . . . . . . . . . . . # . . # . . . . . # . . # . . .
. . . . . . . . . . . . . . . . . # # . . . . . . . # # . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . # . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . # . # . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . # . # . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . # . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . # . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . # . # . . # # . . . . . . . . . . . . . .
. . . . . . . . . . . . # # . . # . # . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . # . . . . . . . . . . . . . .
. . . . . # # . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . # # . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . # # # . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . # # . . . . . # .
. . . . . . . . . . . . . . . . . . . . . . . # # . . . . . # .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . # .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . # . . . # # # . . .
```

The above state is the state where on upon the actions of the rules, there are no further generations and the state remains the same for all the successive iterations.

Since in this project the whole implementation is done in parallel hence the speedup is proportional to the number of processes (**P = 1**)

Limitations and Possible Improvements

The major limitation of con way's game of life is that counting the alive neighbours becomes a hash-lookup or a search operation regardless of whether the code is implemented sequentially or in parallel, which in turn reduces the simulation time.

The parallel implementation can be further improved when executed on a system with higher number of cores which can in-turn handle larger number of processes, thereby reducing the time taken by the initial grid configuration to reach a stable state.

Conclusion

The main conclusion drawn by implementing cellular automation in a parallel fashion are :

- Increase in the speedup of the simulation with the increase in the the number of processes working on the successive generations of the grid.

- **Tp < Ts** and **Ip < Is**
  (Ts = time taken by the grid to reach stable state in sequential implementation)
  (Tp = time taken by the grid to reach stable state in parallel implementation)
  (Ip = number of iterations taken by the grid to reach stable configuration in parallel implementation)
  (Is = number of iterations taken by the grid to reach stable configuration in sequential implementation)

- **Speedup directly proportional to the number of processes**

References

1. **"Conway's Game of Life Cellular Automata" by Andrew Adamatzky (Springer Publications)**
2. **David B. Kirk "Programming Massively Parallel Processors" (for CUDA).**

Link to Project

https://github.com/whiz-Tuhin/Parallel-Game-of-Life