Parallelization of
Conway's Game of Life
DSC 520 – Final Project Report

Prepared By: **Rohan Gonjari**

Student ID: **01987448**

Investigator: **Professor Alfa Heryudono**

University: **University of Massachusetts, Dartmouth, MA - 02747**

I. Abstract

This project deals with the parallel implementation of Conway's Game of Life via the Message Passing Interface technique which uses distributed memory. This project will introduce the reader to Conway's Game of Life which is a popular cellular automaton devised by British Mathematician John Conway in 1970. Cellular automaton is a system in which a set of rules, defined by the user, are applied to cells and their neighbors in a regular grid. The project will also compare runtimes of the Game of Life in its serial implementation and its parallel implementation with four and eight cores.

II. Motivation

Parallel computing is a type of computation in which multiple processes are carried out simultaneously. Larger tasks can be broken down into smaller sub-tasks and these sub-tasks can be distributed amongst cores or workers to run them simultaneously. Parallelization can either be run on shared memory or distributed memory models. We will be employing the distributed memory model, where communication amongst cores occurs using the send and receive messages command in the MPI module. The following factors played a vital role in the parallelization of the Game of Life:

i. Speedup

A parallel implementation of a program will run faster when compared to its serial implementation. The notion of speedup can be used to show the effect on performance using execution times, and it also allows problems to be modelled faster. In this project, we focus on calculating speedups of our various implementations.

ii. Scaling

Scaling is another primary reason as to why one would pursue parallelization. When a higher number of cores or workers are utilized, it greatly increases the computational power. For this project, we increase the scale by increasing the size of the regular girds.

III. Methodology

We pursue this subject matter to record the runtimes, speedups, and efficiencies of our various implementations. We measure speedup of the implementation using the given formula:

$$S_P = T_S / T_P$$

where,

S_P: Speedup using P cores

T_S: Runtime in serial implementation

T_P: Runtime with P cores

P: Number of Cores

We also compare the efficiencies of the parallel implementations with four cores and eight cores. Efficiency of a method can be calculated using the given formula:

$$E_P = S_P / P$$

where,

E_P: Efficiency with P cores

S_P: Speedup using P cores

P: Number of Cores

Upon completion of this project, we attempt to draw conclusions from our findings and determine which implementation is the quickest and most efficient.

IV. MPI Implementation

This project was implemented using the Message Passing Interface (MPI) which uses a distributed memory model. MPI is a message-passing system designed for multiple cores and allows them to communicate with each other with the help of MPI commands.

The following rules are defined for our implementation of the Game of Life:

i. If an alive cell has less than two neighbors, it will be declared dead.
ii. If an alive cell has more than three neighbors, it will be declared dead.
iii. If a dead cell has exactly three neighbors, it will be declared alive.

The serial and parallel program define an initial regular grid. In the serial implementation, the new state of the grid is governed by its neighbors and a set of rules defined by the user.

In our parallel implementation, the rows of the regular grid are divided equally amongst the number of cores available to create sub-grids. We use MPI to share information about the current state of the sub-grids and we share this information with the neighboring sub-grids. With the help of this information, we then calculate the new state for each sub-grid with the help of our pre-defined rules. Once the new state for each sub-grid is generated, we gather the data in each sub-grid and print out the newly generated grid.

This marks the end of one generation. To begin computation for a new generation, we share information of the sub-grids again with its neighboring sub-grids and generate the next generation.

V. Results

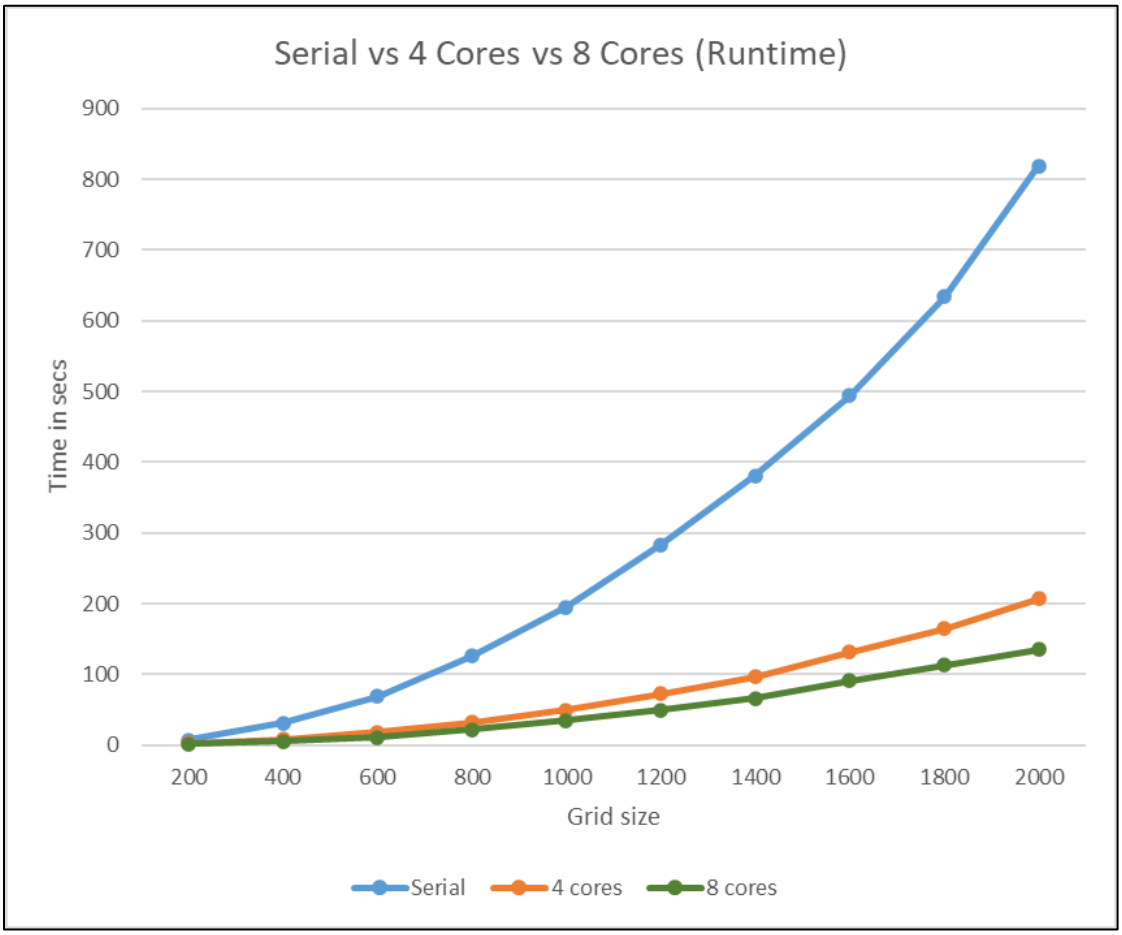
Conway's game of life was run on regular grids of various sizes to gain a better understanding of the speedup and efficiency. The following configurations were implemented:

i. Serial implementation
ii. Parallel implementation with 4 cores
iii. Parallel implementation with 8 cores

The execution times were first recorded for each implementation with different grid sizes. The results are listed in the table below:

Grid N x N	Serial runtime(secs)	MPI 4 cores runtime(secs)	MPI 8 cores runtime (secs)
200	7.308	2.204	1.375
400	30.892	8.201	5.073
600	68.701	18.343	11.177
800	126.213	32.498	21.791
1000	194.983	50.008	34.456
1200	283.565	72.995	49.867
1400	380.565	96.602	66.38
1600	493.587	131.249	90.887
1800	633.455	164.906	113.473
2000	819.171	206.813	135.28

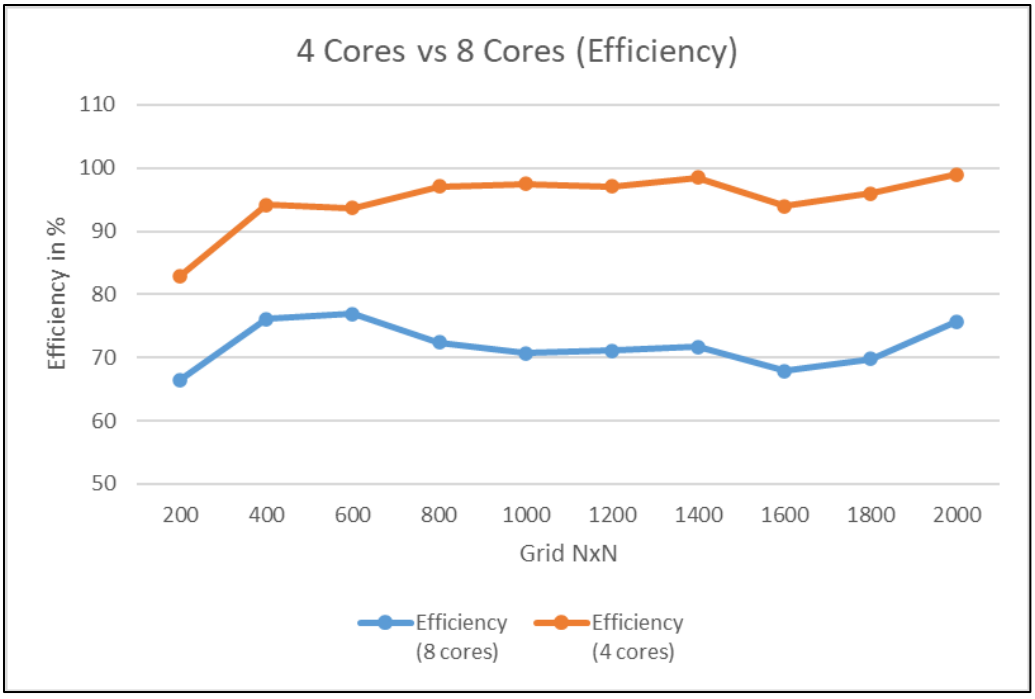
The above table of runtimes is graphed in the chart below to provide a visual understanding:



With the execution times, the speedups, and efficiencies for the four cores and eight cores implementation were calculated to determine which method is more efficient. The calculations were recorded in the table below:

4 Cores		8 Cores	
Speedup (secs)	**Efficiency in %**	**Speedup (secs)**	**Efficiency in %**
3.315	82.894	5.314	66.436
3.766	94.171	6.089	76.118
3.745	93.633	6.146	76.833
3.883	97.092	5.791	72.399
3.899	97.475	5.658	70.736
3.884	97.117	5.686	71.080
3.939	98.487	5.733	71.664
3.760	94.017	5.430	67.884
3.841	96.032	5.582	69.780
3.960	99.023	6.055	75.692

The above table of efficiencies is graphed in the chart below to provide a visual understanding:



VI. Conclusions

From our results, the following conclusions can be drawn upon with respect to speedup and efficiency:

i. Parallel vs Serial

Time taken by the parallel implementations is a lot less than the time taken for serial implementation. We can say that parallelization effectively reduces the execution time of the program while providing the same output. This directly increases the efficiency of the machine.

ii. Four cores vs Eight cores

With regards to the number of cores, the quad core configuration is more efficient than the octa core configuration. This is possible due to the time it takes for workers to communicate with each other. Hence, with an increase in the number of cores, communication time will increase, which will indirectly reduce the efficiency slightly.

VII. Real-world Applications

Parallelization of a process holds no meaning unless it can be applied in the real world. Now that we have parallelized the Game of Life, we can apply the parallelization to other cellular automaton systems. These systems are used to model a wide variety of natural phenomenon such as:

i. Used to simulate brain tumor growth by generating a 3-D scan of the brain and advancing cellular growth over a period of time based on a set of rules.
ii. In ecology, cellular automaton is at times used to model the behavior of certain species and how they compete for resources.
iii. Cellular automaton is used to design public keys for cryptosystems. It is also used in the fields of cognitive science, hydrodynamics, agriculture etc.

VIII. References

[1] "Conway's Game of Life Cellular Automata" by Andrew Adamatzky (Springer Publications).

[2] Program for Conway's Game Of Life (https://www.geeksforgeeks.org/program-for-conways-game-of-life/).

[3] Asanovic K, The Landscape of Parallel Computing Research: A View from Berkley.

[4] Jauntbox/paralife (https://github.com/Jauntbox/paralife)

[5] MPI for Python (https://mpi4py.readthedocs.io/en/stable/)

IX. Appendix

Code for Serial implementation of Conway's Game of Life:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time
from numpy.core.fromnumeric import repeat

N = 200
ON = 255
OFF = 0
vals = [ON, OFF]
grid = np.random.choice(vals, N*N, p=[0.6, 0.4]).reshape(N, N)

def game_of_life(data):
  global grid
  #Copy grid to a new grid
  newGrid = grid.copy()
  for i in range(N):
    for j in range(N):
      #Taking toroidal boundary conditions
      total = (grid[i, (j-1)%N] + grid[i, (j+1)%N] +
               grid[(i-1)%N, j] + grid[(i+1)%N, j] +
               grid[(i-1)%N, (j-1)%N] + grid[(i-1)%N, (j+1)%N] +
               grid[(i+1)%N, (j-1)%N] + grid[(i+1)%N, (j+1)%N])/255
      #Rules of the game
      if grid[i, j]  == ON:
        if (total < 2) or (total > 3):
          newGrid[i, j] = OFF
      else:
        if total == 3:
          newGrid[i, j] = ON
  #Updating grid
  mat.set_data(newGrid)
  grid = newGrid
  return mat

fig, ax = plt.subplots()
mat = ax.matshow(grid)
ani = animation.FuncAnimation(fig, game_of_life, frames=5, interval=0,
                              save_count=50, repeat=False)
plt.show()
```

Code for Parallel implementation of Conway's Game of Life using MPI:

```
import numpy
import sys
from matplotlib import pyplot as plt
from mpi4py import MPI
import time

prob = [0.6]
COLS = 200
ROWS = 200
generations = 100

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
stat = MPI.Status()

subROWS = ROWS//size+2

def msgUp(subGrid):
        # Sends and Recvs rows with Rank+1
        comm.send(subGrid[subROWS-2,:],dest=rank+1)
        subGrid[subROWS-1,:]=comm.recv(source=rank+1)
        return 0

def msgDn(subGrid):
        # Sends and Recvs rows with Rank-1
        comm.send(subGrid[1,:],dest=rank-1)
        subGrid[0,:] = comm.recv(source=rank-1)
        return 0

def newGenetation(subGrid):
    intermediateG = numpy.copy(subGrid)
    for ROWelem in range(1,subROWS-1):
        for COLelem in range(1,COLS-1):
            neighbour_sum = ( subGrid[ROWelem-1,COLelem-1]+subGrid[ROWelem-
1,COLelem]+subGrid[ROWelem-1,COLelem+1]
                              +subGrid[ROWelem,COLelem-
1]+subGrid[ROWelem,COLelem+1]
                              +subGrid[ROWelem+1,COLelem-
1]+subGrid[ROWelem+1,COLelem]+subGrid[ROWelem+1,COLelem+1] )
            if subGrid[ROWelem,COLelem] == 1:
                if neighbour_sum < 2:
                    intermediateG[ROWelem,COLelem] = 0
                elif neighbour_sum > 3:
                    intermediateG[ROWelem,COLelem] = 0
```

```
  else:
                    intermediateG[ROWelem,COLelem] = 1
            if subGrid[ROWelem,COLelem] == 0:
                if neighbour_sum == 3:
                    intermediateG[ROWelem,COLelem] = 1
                else:
                    intermediateG[ROWelem,COLelem] = 0
    subGrid = numpy.copy(intermediateG)
    return subGrid

for p in prob:
    N=numpy.random.binomial(1,p,size=(subROWS+2)*COLS)
    subGrid = numpy.reshape(N,(subROWS+2,COLS))

    # BC for all ranks.
    subGrid[:,0] = 0
    subGrid[:,-1] = 0

tic = time.time()
#compute new grid and pass rows to neighbors
oldGrid=comm.gather(subGrid[1:subROWS-1,:],root=0)
for i in range(1,generations+1):
    subGrid = newGenetation(subGrid)
#exhange edge rows for next interation
    if rank == 0:
        msgUp(subGrid)
    elif rank == size-1:
        msgDn(subGrid)
    else:
        msgUp(subGrid)
        msgDn(subGrid)
    newGrid=comm.gather(subGrid[1:subROWS-1,:],root=0)
    if rank == 0:
        result= numpy.vstack(newGrid)
        print('Generation {} running ...'.format(i))
        plt.imsave('newstate/temp'+str(i)+'.jpg',result)
        plt.show()
toc = time.time()
print(toc-tic)
```

~