Header:
Author: Crissy Hendrickson
Title: Game of Life
Group Members: Benjamin Kasavan

Analysis of the Problem:

Game of Life was first introduced by Conway and is a cellular automation. It is both universal and undecidable which means an arbitrary output is generated for an arbitrary input, and it is impossible to solve the model without simulating it. Specifically, the game starts with a square grid filled with cells that are either dead or alive. For every time step taken, each cell interacts with its eight neighbors according to three transition rules to determine if the cell lives or dies. The transition rules are as follows: a living cell with fewer than two living neighbors dies, a living cell with two or three living neighbors lives, a living cell with more than three living neighbors dies, a dead cell with three living neighbors becomes alive, and all other dead cells remain dead. In Game of Life many different patterns can arise with the living cells including still lifes which don't move, oscillators which alternate between different patterns, and spaceships which move across the grid. The system can overall be simulated with sequential or parallel code, the latter of which uses a pool of workers. It can be expected that as more workers are used to simulate the system, the simulation time will initially decrease but then increase as the cost of communication rises.

Model Design:.

Computational models are simplified representations of phenomena, systems, or processes. They are used to help us study and understand different aspects of life and can lead to predictions involving responses in un-tested scenarios. Often because of certain assumptions and simplifications, they are much easier to work with than the systems they represent and can point to new insight and comprehension. To translate a problem into a computational model there are five main steps: analysis of a problem, model development and formulation, model implementation, model verification, and model interpretation. The first step, analysis of a problem, involves determining what the question and unknowns are and what the answer would look like. The second step, model development and formulation, includes gathering relevant data, making assumptions, determining variable relationships and units, and writing down specific equations or rules. Model implementation involves writing a program in a computer language to solve the problem while model verification involves testing special cases whose answers are known to confirm the solution is accurate and the program does what it is supposed to do. Finally, model interpretation includes answering the main question sometimes with figures or tables and discussing more about the solution: if it was expected, what it means and how it could change if the assumptions are relaxed.

In the model for Game of Life, several assumptions were made before the model was implemented. The model was assumed to be dynamic since it depends on time and stochastic because the solution involves an elements of chance and may be different even under the same conditions. It is also assumed to be a spatially extended model because the solution is given by a field that is different at different points in space. Because time changes in specific well separated steps, time is considered to be discrete. Furthermore, space is considered to be discrete because a lattice is used to specify it. The boundary conditions were chosen to be absorbing and the size of the lattice was chosen to be 120x120. That particular size was chosen because it is easily divided by many different numbers and the size

needs to be a multiple of the number of workers so each lattice can be divided evenly in the parallel code.

In the model development, several Python functions were used to help find the solution. The specific code is described in the following section.

Model Solution:

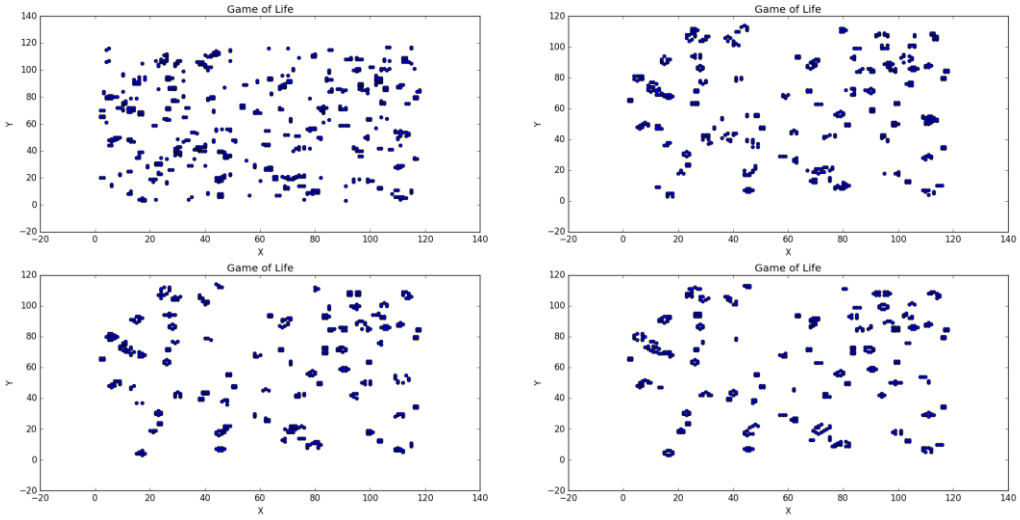
To find the solution, the code at the end of the report was run in Python. First, the variables were initialized so the size of the grid, the empty cells at the edge of each lattice, and the probability of each element starting with a living cell could be defined. Additionally, the number of workers and the amount of discrete time units to advance in were defined. Booleans were also used so one could specify if the simulation used sequential or parallel code and if it had animation or not. Next a grid was created and filled randomly with zeros and ones using the random number generator in numpy and two for loops. The grid also had absorbing boundary conditions which were established by setting everything beyond the initial boundary equal to zero. To verify the solution, a grid was also filled with various oscillators, still lifes, and spaceships to see if the cells behaved as expected.

The sequential code was then written using for loops and an array called timeEvolution which kept track of the state of each cell in the lattice at each moment in time. The next state of each cell was determined using the transition rules and a copy of the timeEvolution array at the previous time step. Additionally, the time sequential processing took was calculated using time.time(). If it was desired, a movie of the simulation could be created using animation from matplotlib and time lapsed images of the simulation could be created.

The parallel code included a part which divided the overall grid and another part which took each new sub grid and moved it forward in time. To divide the overall grid first the number of partitions and their length was determined. Then a new array was created which was a copy of timeEvolution at the previous time step and the boundary conditions were inserted and the data was partitioned using np.insert(). To partition the overall grid into sub grids, two for loops were used and lists collected the information to be distributed over processes. Then using pool from multiprocessing, the tasks were divided among the workers and sent to a function called forward which took each sub grid and moved it forward in time according to the transition rules. The data was then returned and complied in the timeEvolution array. Additionally, the time parallel processing took was calculated using time.time(). If it was desired, a movie of the simulation could be created using animation from matplotlib and time lapsed images of the simulation could be created. The code was run with different amounts of workers and the time it took to complete was recorded and then put into a graph using plt.plot. so the speed versus number of workers could be determined.

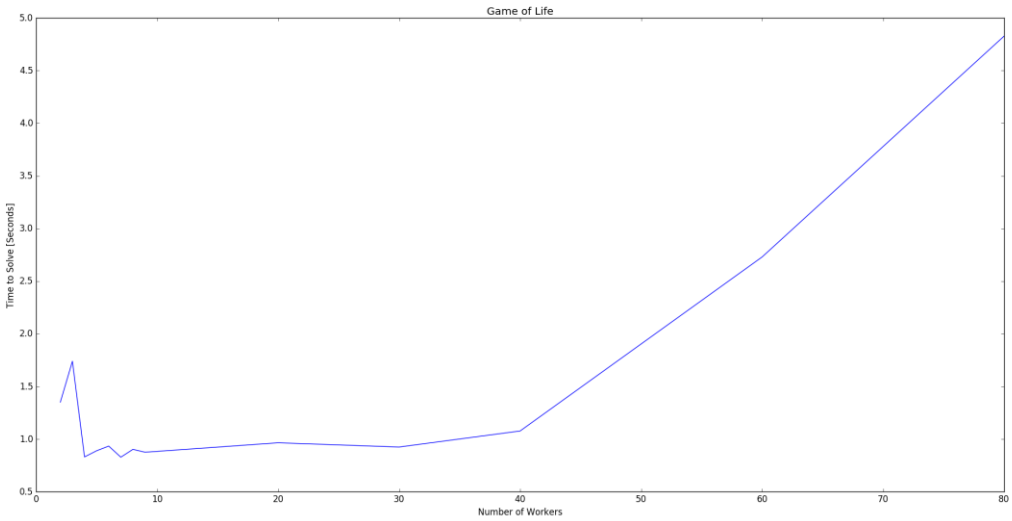
Results, Verification, and Conclusion:

The following graphs show a time lapse of Game of Life at four consecutive moments in time.



The top left image was taken at the first time step, and the top right image was taken at the second time step. The bottom left image was taken at the third time step and the bottom right image was taken at the fourth time step. In Game of life there is an example of a block near (100,64) a blinker near (58,28), a beehive near (45,7) among other things.

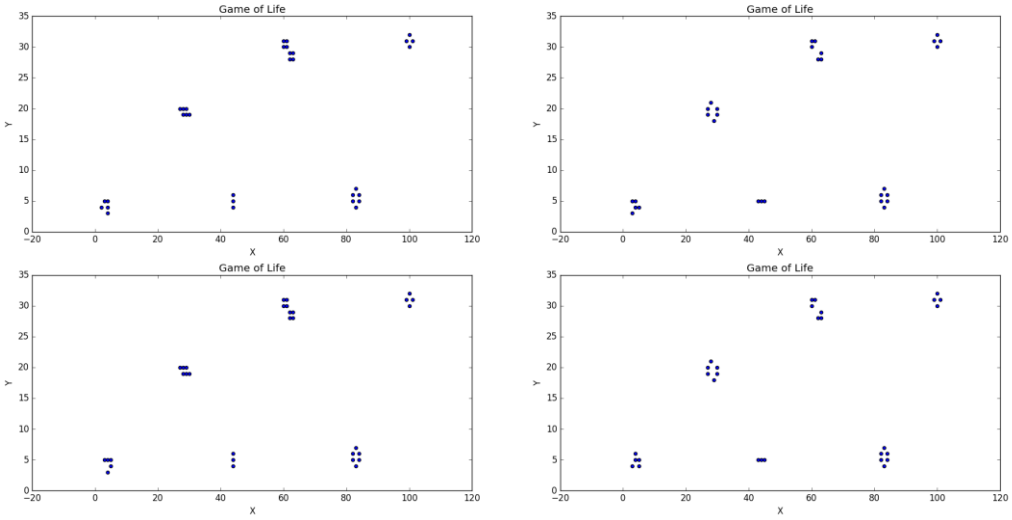
A graph was also made which showed the speed versus number of workers.



As shown by the graph above, the speed initially decreased when more workers were added but eventually increased again. The behavior was expected because as more workers are added the cost of communication rises so eventually it is no longer faster to use more workers. Next time to improve the model, the parallel code needs to be modified so it can run on a windows computer. Currently, the parallel code only works on a MAC so to make the model more accessible to everyone it needs to be altered to also work on a windows computer.

To verify the code worked, a grid was filled with various oscillators, still lifes, and spaceships to see if the cells behaved as expected.

As



shown by the time lapse above the toad, beacon, tub, glider, blinker and beehive all behave as expected so the solution is verified.

Code:

```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 20 14:28:38 2017

@author: Crissy
"""

import numpy as np
import numpy.random as nprnd
import time
from multiprocessing import Pool
from multiprocessing import cpu_count
from functools import partial
import matplotlib.pyplot as plt
from matplotlib import animation



##############################
#Variable Initiatlizationthoin
##############################
gridSize = 120#multiple of number of workers, 5!
gridBuffer = 2 #how many empty cells on each edge of the lattice.
initProb = 0.1 #probability of each element starting with a live cell
workers = 4
timeStep = 10 #how many discrete time units to advance in
parallel = False
sequential = True
animate = True
```

```
##############################
#End  of  Initiatlizationthoin
##############################

#%%
#First creating the grid and filling it randomly with zeros and ones.

grid = nprnd.random(2*[gridSize])
for i in range(gridSize):
    for j in range(gridSize):
        if i < gridBuffer or i > (gridSize-gridBuffer) or \
        j < gridBuffer or j > (gridSize - gridBuffer):
            grid[i, j] = 0
            #everything beyond the initnial boundary is dead
        if grid[i, j] > (1-initProb):
            grid[i, j] = 1
        else:
            grid[i, j] = 0

#verification
#grid=np.zeros(2*[gridSize])
#
##toad
#grid[28,19]=1
#grid[29,19]=1
#grid[30,19]=1
#grid[27,20]=1
#grid[28,20]=1
#grid[29,20]=1
#
##beacon
#grid[60,30]=1
#grid[61,30]=1
#grid[60,31]=1
#grid[61,31]=1
#grid[62,29]=1
#grid[62,28]=1
#grid[63,29]=1
#grid[63,28]=1
#
##tub
#grid[100,30]=1
#grid[101,31]=1
#grid[99,31]=1
#grid[100,32]=1
#
##glider
#grid[4, 3] = 1
#grid[4, 4] = 1
#grid[4, 5] = 1
#grid[3, 5] = 1
#grid[2, 4] = 1
#
```

```
##blinker
#grid[44, 4] = 1
#grid[44, 5] = 1
#grid[44, 6] = 1
#
##beehive
#grid[82, 5] = 1
#grid[82, 6] = 1
#grid[84, 5] = 1
#grid[84, 6] = 1
#grid[83, 7] = 1
#grid[83, 4] = 1


numNeighbours = np.zeros(2*[gridSize])
#Array which will store how many neighbours each cell has.
#%%
##############################
#Sequential Solution
##############################
if sequential:

    timeEvolution = np.zeros([timeStep, gridSize, gridSize])
    ts = time.time()
    for k in range(timeStep):
        if k == 0:
            timeEvolution[k] = np.copy(grid)
            continue
        for i in range(gridSize-1):
            for j in range(gridSize-1):
                numNeighbours = 0
                grid = np.copy(timeEvolution[(k-1)])
                if grid[i-1, j] == 1:
                    numNeighbours += 1
                if grid[i, j-1] == 1:
                    numNeighbours += 1
                if (grid[i-1, j-1] == 1):
                    numNeighbours += 1
                if (grid[i+1, j] == 1):
                    numNeighbours += 1
                if (grid[i, j+1] == 1):
                    numNeighbours += 1
                if (grid[i+1, j+1] == 1):
                    numNeighbours += 1
                if (grid[i-1, j+1] == 1):
                    numNeighbours += 1
                if (grid[i+1, j-1] == 1):
                    numNeighbours += 1

                if numNeighbours < 2:
                    timeEvolution[k, i, j] = 0
                elif numNeighbours == 2 and grid[i, j] == 1:
                    timeEvolution[k, i, j] = 1
```

```
                elif numNeighbours == 2 and grid[i, j] == 0:
                    timeEvolution[k, i, j] = 0
                elif numNeighbours == 3:
                    timeEvolution[k, i, j] = 1
                elif numNeighbours >3:
                    timeEvolution[k, i, j] = 0
                else:
                    #if none of the other cases worked,
                    #then something went wrong.
                    timeEvolution[k, i, j] = -5

    print('Sequential processing took {}s'.format(time.time() - ts))

#%%
##############################
#Animate it!
##############################
if animate and sequential:
    nonzero=[]
    x=[]
    y=[]

    for k in range(timeStep):
        nonzero = np.transpose(np.nonzero(timeEvolution[k]))
        N=np.size(nonzero)
        x.append(nonzero[0:N:1,0])
        y.append(nonzero[0:N:1,1])

    ax = plt.axes(xlim=(0 ,gridSize), xlabel='X', ylim=(0, gridSize), \
                  ylabel='Y', title='Game of Life')
    fig = plt.figure()
    line, = ax.plot([], [], 'ro', markersize=2)

    # initialization function: plot the background of each frame
    def init():
        line.set_data([], [])
        return line,


    # animation function.  This is called sequentially
    def animate(i):
        xD = x[i]
        yD = y[i]
        line.set_data(xD, yD)
        return line,

    # call the animator.  blit=True means only
    #re-draw the parts that have changed.
    anim = animation.FuncAnimation(fig, animate, init_func=init,
                                   frames=timeStep, interval=500, blit=True)

    #Time lapse
```

```
    t1=1
    t2=2
    t3=3
    t4=4

    plt.subplot(2,2,1)
    plt.scatter(x[t1], y[t1])
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Game of Life')
    plt.subplot(2,2,2)
    plt.scatter(x[t2], y[t2])
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Game of Life')
    plt.subplot(2,2,3)
    plt.scatter(x[t3], y[t3])
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Game of Life')
    plt.subplot(2,2,4)
    plt.scatter(x[t4], y[4])
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Game of Life')

#%%
##############################
#forward function
##############################
#takes each grid and moves it forward after it has been divided
def forward(gridS):
    """  return: grids after one time step """
    tmp = np.zeros((gridS.shape[0] - 2, gridS.shape[1] - 2))
    for i in np.arange(1,gridS.shape[0]-1):
        for j in np.arange(1,gridS.shape[1]-1):
            numNeighbours = 0
            if gridS[i-1, j] == 1:
                numNeighbours += 1
            if gridS[i, j-1] == 1:
                numNeighbours += 1
            if gridS[i-1, j-1] == 1:
                numNeighbours += 1
            if gridS[i+1, j] == 1:
                numNeighbours += 1
            if gridS[i, j+1] == 1:
                numNeighbours += 1
            if gridS[i+1, j+1] == 1:
                numNeighbours += 1
            if gridS[i-1, j+1] == 1:
                numNeighbours += 1
            if gridS[i+1, j-1] == 1:
                numNeighbours += 1
```

```
            if numNeighbours < 2:
                tmp[i-1, j-1] = 0
            elif numNeighbours == 2 and grid[i, j] ==1:
                tmp[i-1, j-1] = 1
            elif numNeighbours == 2 and grid[i, j] ==0:
                tmp[i-1, j-1] = 0
            elif numNeighbours == 3:
                tmp[i-1, j-1] = 1
            elif numNeighbours >3:
                tmp[i-1, j-1] = 0
            else:
                #if none of the other cases worked, then something went wrong.
                tmp[i-1, j-1] = -5
    return tmp

##############################
#Parallel Solution
##############################

#Boundary conditions of each sub lattice: Each edge takes the
#value of the edge of the adjacent lattice.
#make a function. "Take this function and apply to this data".
#Give it lattice with boundary conditions, it returns the lattice without them.
#func = partial(diffusion, D, dt, dx) = makes sure each time the diffucion
#function gets called it has the same D, dt, dx values and only takes a
#new grid argument.
#divides the grids to be given to forward function
if parallel:
    print("Workers: " + str(workers))
    timeEvolution = np.zeros([timeStep, gridSize, gridSize])
    p = Pool(workers)
    gridsize = grid.shape

    # define how many partitions of grid in x and y direction and their length
    (nx, ny) = (int(workers / 2), 2)
    lx = int(gridsize[0] / nx)
    ly = int(gridsize[1] / ny)

    ts = time.time()  # measure computation time
    for t in range(timeStep):
        if t ==0:
            timeEvolution[t] = np.copy(grid)
            continue
        grid = np.copy(timeEvolution[t-1])
        data = []
        # prepare data to be distributed among workers
        # 1. insert boundary conditions and partition data
        grid = np.insert(grid, (0, gridsize[0]), grid[(0, -1), :], axis=0)
        grid = np.insert(grid, (0, gridsize[1]), grid[:, (0, -1)], axis=1)
        # partition into subgrids
        for i in range(nx):
            for j in range(ny):
```

```
                # subgrid
                subg = grid[i * lx + 1:(i+1) * lx + 1, j * ly +\
                            1:(j+1) * ly + 1]
                # upper boundary
                subg = np.insert(subg, 0, grid[i * lx, j * ly + \
                                               1:(j+1) * ly + 1],
                                 axis=0)
                # lower boundary
                subg = np.insert(subg, subg.shape[0],
                                 grid[(i+1) * lx + 1, j * ly + 1:(j+1)\
                                      * ly + 1],
                                 axis=0)
                # left boundary
                subg = np.insert(subg, 0, grid[i * lx:(i+1) * lx + 2, j * ly],
                                 axis=1)
                # right boundary
                subg = np.insert(subg, subg.shape[1],
                                 grid[i * lx:(i+1) * lx + 2, (j+1) * ly + 1],
                                 axis=1)
                # collect subgrids in list to be distributed over processes
                data.append(subg)
        # 2. divide among workers
        results = p.map(forward, data)
        #gives the function to forward and returns the data
        grid = np.vstack([np.hstack((results[i * ny:(i+1) * ny]))\
                          for i in range(nx)])
        timeEvolution[t] = np.copy(grid)


    print('Parallel processing took {}s'.format(time.time() - ts))

#%%
##############################
#Animate it!
##############################
if animate and parallel:
    nonzero=[]
    x=[]
    y=[]

    for k in range(timeStep):
        nonzero = np.transpose(np.nonzero(timeEvolution[k]))
        N=np.size(nonzero)
        x.append(nonzero[0:N:1,0])
        y.append(nonzero[0:N:1,1])

    ax = plt.axes(xlim=(0 ,gridSize), xlabel='blah', ylim=(0, gridSize))
    fig = plt.figure()
    line, = ax.plot([], [], 'ro', markersize=2)

    # initialization function: plot the background of each frame
    def init():
        line.set_data([], [])
```

```
        return line,


    # animation function.  This is called sequentially
    def animate(i):
        xD = x[i]
        yD = y[i]
        line.set_data(xD, yD)
        return line,

    # call the animator.  blit=True means only
    #re-draw the parts that have changed.
    anim = animation.FuncAnimation(fig, animate, init_func=init,
                                   frames=timeStep, interval=500, blit=True)

    #Time lapse

    t1=1
    t2=2
    t3=3
    t4=4

    plt.subplot(2,2,1)
    plt.scatter(x[t1], y[t1])
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Game of Life')
    plt.subplot(2,2,2)
    plt.scatter(x[t2], y[t2])
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Game of Life')
    plt.subplot(2,2,3)
    plt.scatter(x[t3], y[t3])
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Game of Life')
    plt.subplot(2,2,4)
    plt.scatter(x[t4], y[t4])
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Game of Life')

#%%
################################
#Speed Vs. Number of Workers
################################
#for a size of 120
Worker_Number=[2,3,4,5,6,7,8,9,10,20,30,40,60,80]
time_to_completion=[1.351,1.740,.831,.8895,.934,.8286,.9028,.8758,.8837,\
                    .9652,.925,1.0774,2.73,4.8277]

plt.figure()
```

```
plt.plot(Worker_Number,time_to_completion)
plt.xlabel('Number of Workers')
plt.ylabel('Time to Solve [Seconds]')
plt.title('Game of Life')
```