A Technical Report on

Simulating the Spread of Forest Fires Adopting the Cellular Automaton Model

Table of Contents

- **Abstract**
- **Introduction**
- **Aim and Objectives**
- **Methodology**
    - ☐ **Setup**
    - ☐ **Cellular Automation Method**
    - ☐ **Model Formulation**
    - ☐ **Methods and implementation**
- **Result and Evaluation**
- **Conclusion**
- **References**

Abstract

The spread of forest fires is a major threat to both humans and animals, and predicting the direction of a fire's spread can be difficult. This report presents a simulation model based on Cellular Automaton that accurately simulates the spread of forest fires. The model takes into account various stochastic parameters, including the probability of a tree being immune to burning and the probability of lightning. The model was implemented in Python, using the Von Neuman and Moore Neighborhood principles to determine transition rules for simulating the spread of the fire. Numba and Pool Multiprocessing approach was adopted to parallelize the sequential implementation, which allowed us to measure the runtime execution of each implementation. Our results show that the Moore Neighborhood parallel implementation outperformed the Von Neuman sequential implementation in terms of runtime. This model can help improve fire management strategies and prevent the loss of lives and properties to wildfires.

Introduction

The spread of fire in a forest is a major threat to both humans and animals that are the habitat of a forest and the direction of the spread of a fire is most times unpredictable which has made it a problem to make proper decisions to prevent loss of lives and properties to wildfires. So, therefore, scientists have come up with various methodologies to simulate and predict the spread of fires in the forest and this can also help in improving fire management strategies.

The motivation behind this algorithm is to simulate the spread of fire within a given domain. This model can be simulated using different probabilistic parameter variables for the prediction such probability of a tree being immune to burning, the probability of lightning etc.

The objective of this algorithm is to create a model based on Cellular Automaton that accurately simulates the spread of forest fires. The model will be based on the principles of cellular automation and will take into account various parameters which include the probabilistic parameters listed above.

Aim

The aim of this coursework is to present an algorithm for simulating the spread of forest fires using Cellular Automaton, specifically focusing on the implementation of both Sequential and parallel techniques for improving the computational efficiency of the simulation. The report aims to provide a detailed explanation of the model formulation, setup, and implementation, along with the results and evaluation of the algorithm.

Objectives

The objectives of this coursework implementation are as follows:

1. To implement the sequential spread of forest fire algorithm using the defined transition rules.
2. Optimize the sequential implementation using the parallelization technique.
3. To present the results and evaluation of the algorithm, including the comparison of runtime execution between sequential and parallel implementations for different grid sizes, and the assessment of the computational efficiency of the algorithm.
4. To recommend further implementation to be done to improve the architecture of the algorithm and also the runtime complexity.

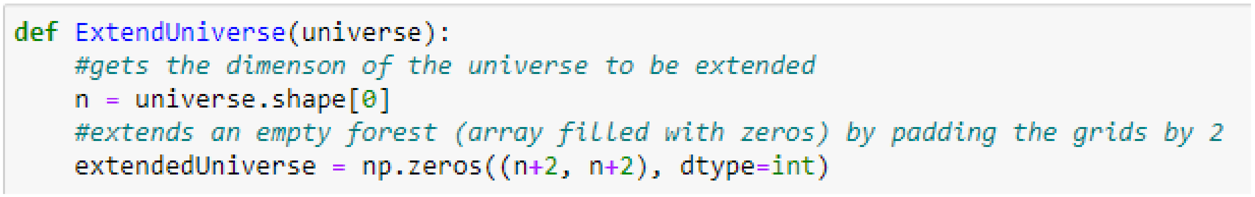
Methodology

Setup: The setup of the model implementation was done in a Python environment by writing the Python programming language in a Jupyter notebook and different necessary libraries were used in the course of this model implementation such as numpy, multiprocessing and matplotlib for graphical user interface interaction for the simulation.

Cellular Automation Method: The cellular automation method adopted for this simulation model is a 2-Dimensional model which has the capability of simulating 2D models of any grid size whereby the cells of this grid are initialized with zeros. The transition of these cell values is determined based on the pre-defined transition rules for simulating the spread of the fire. The transition rules are based on the n * n grid cell values 0, 1, or 2 indicating an empty cell, a cell with a non-burning tree, or a cell with a burning tree, respectively.
The status of each cell (Spread of fire) is determined at every iteration by validating against the pre-defined stochastic parameters and depending on the neighbouring cell values with the aid of Vonn Neuman or Moore Neighborhood principles.

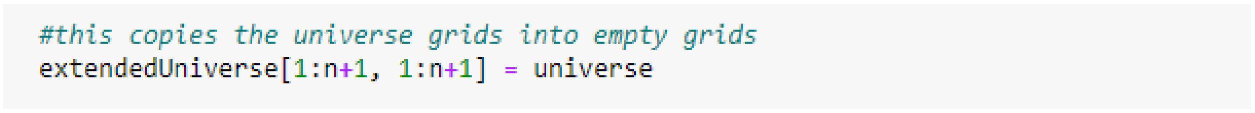
Model Formulation: The "*InitializeUniverse*" function is the entry point of the algorithm and initializes an array (forest or universe) of n * n size with zeros. This 2-Dimensional array(forest) is looped through while randomly assigning trees and setting some of the trees on fire based on the given probabilistic parameters.

The "*ExtendUniverse*" function extends the forest grid to include periodic boundary conditions. This was achieved by getting the dimension of the array (forest) to be extended and then extending an empty forest (array initialized with zeros) by 2 extra rows and columns.



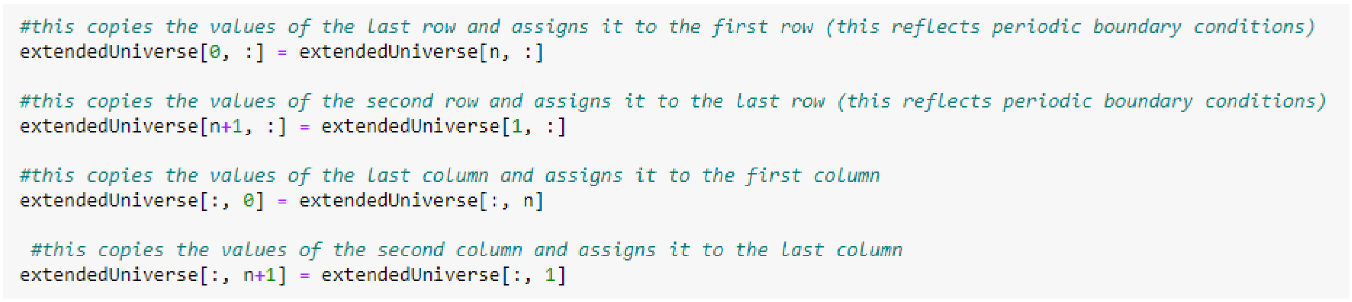
```
def ExtendUniverse(universe):
    #gets the dimenson of the universe to be extended
    n = universe.shape[0]
    #extends an empty forest (array filled with zeros) by padding the grids by 2
    extendedUniverse = np.zeros((n+2, n+2), dtype=int)
```

Then, The burning forest is then copied into the empty forest such that its boundaries are bounded by empty rows and columns to the top and bottom, to the right and left.



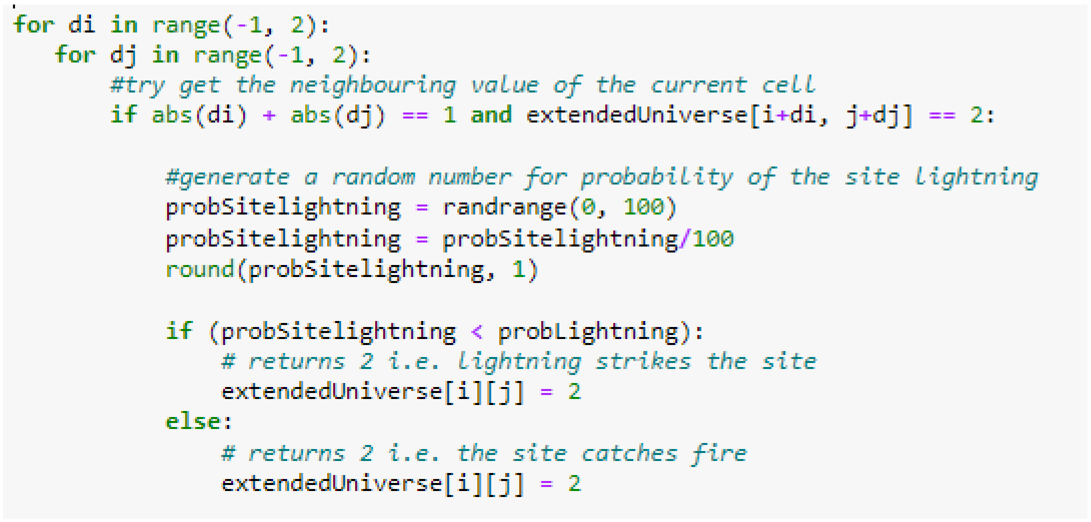
```
#this copies the universe grids into empty grids
extendedUniverse[1:n+1, 1:n+1] = universe
```

Then the periodic boundary conditions were further implemented by copying cell values of the last row into the first row and copying the second row into the last row. The same thing also applies to the columns as well to help us simulate an infinite lattice similar to the topology of a torus.



```python
#this copies the values of the last row and assigns it to the first row (this reflects periodic boundary conditions)
extendedUniverse[0, :] = extendedUniverse[n, :]

#this copies the values of the second row and assigns it to the last row (this reflects periodic boundary conditions)
extendedUniverse[n+1, :] = extendedUniverse[1, :]

#this copies the values of the last column and assigns it to the first column
extendedUniverse[:, 0] = extendedUniverse[:, n]

 #this copies the values of the second column and assigns it to the last column
extendedUniverse[:, n+1] = extendedUniverse[:, 1]
```

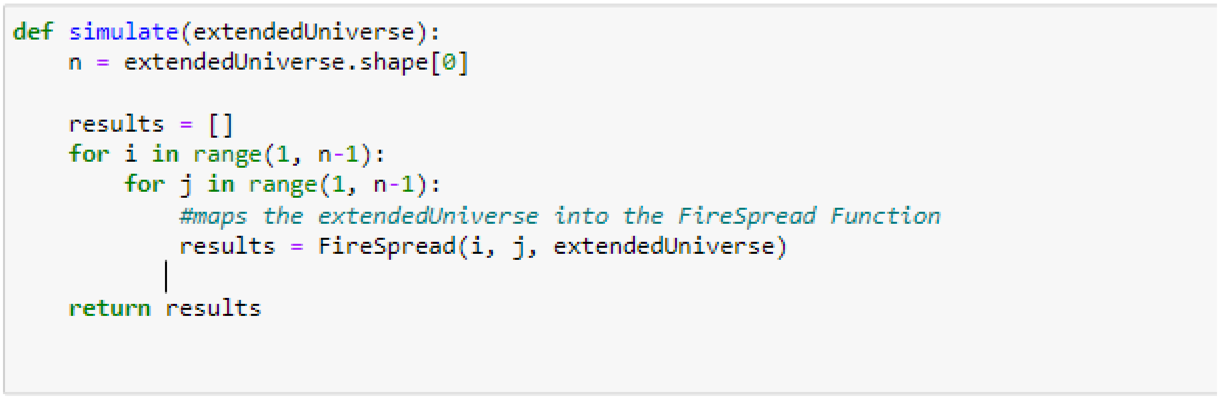
The "FireSpread" function simulates the spread of fire by updating the state of each cell according to certain transition rules (Von Neuman Neighbourhood). If a cell is empty or has a non-burning tree, there is a chance that it will catch fire based on the "probBurning" parameter. If a cell is a burning tree, it will become empty. If a non-burning tree is adjacent to a burning tree, there is a chance that it will catch fire based on the "probLightning" parameter.



```python
for di in range(-1, 2):
    for dj in range(-1, 2):
        #try get the neighbouring value of the current cell
        if abs(di) + abs(dj) == 1 and extendedUniverse[i+di, j+dj] == 2:

            #generate a random number for probability of the site lightning
            probSitelightning = randrange(0, 100)
            probSitelightning = probSitelightning/100
            round(probSitelightning, 1)

            if (probSitelightning < probLightning):
                # returns 2 i.e. lightning strikes the site
                extendedUniverse[i][j] = 2
            else:
                # returns 2 i.e. the site catches fire
                extendedUniverse[i][j] = 2
```

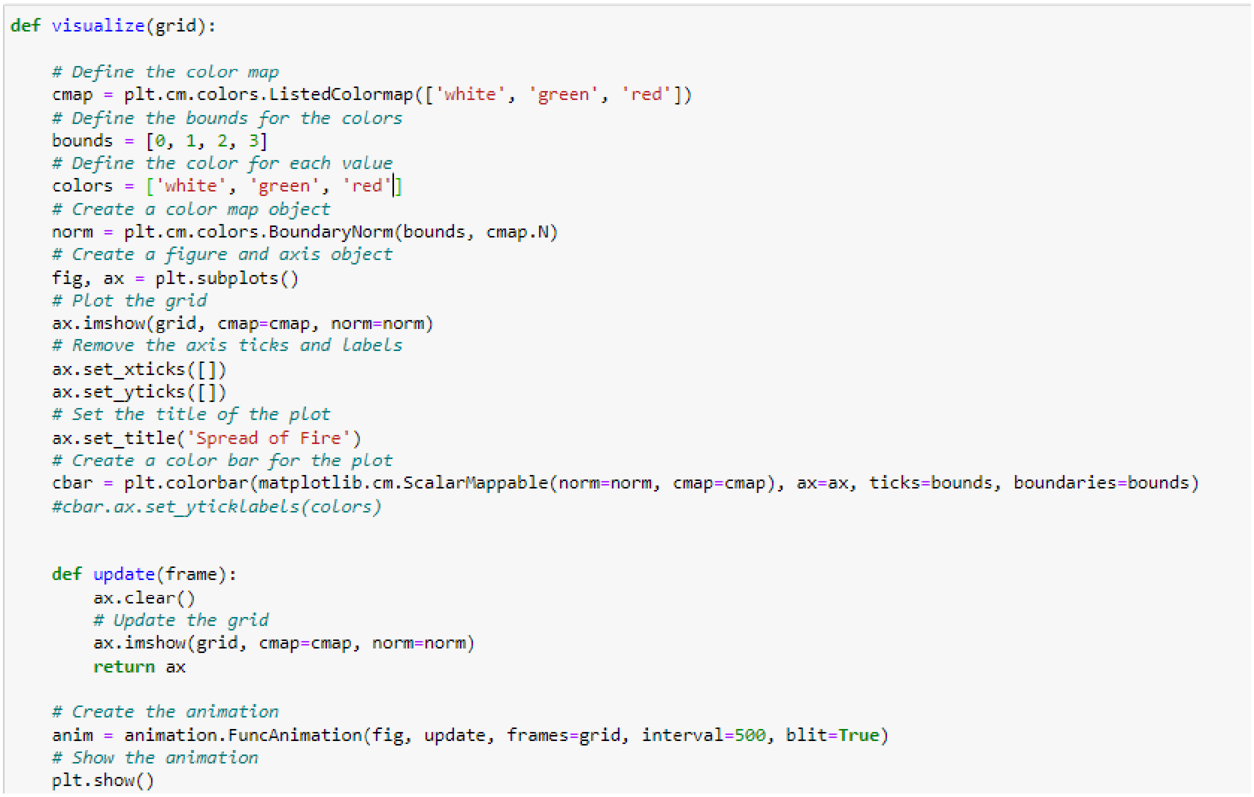
The screenshot of the algorithm above illustrates the Von Neumann Neighbourhood principle was used as the transition rule for the spread of the forest fire.

The "simulate" function uses the "*FireSpread*" function to simulate the spread of fire in the forest. This spread of fire is carried out sequentially.



```python
def simulate(extendedUniverse):
    n = extendedUniverse.shape[0]

    results = []
    for i in range(1, n-1):
        for j in range(1, n-1):
            #maps the extendedUniverse into the FireSpread Function
            results = FireSpread(i, j, extendedUniverse)

    return results
```

The “visualize” function contains implementation to visualize the animated simulated model using the Python library called Matplotlib.

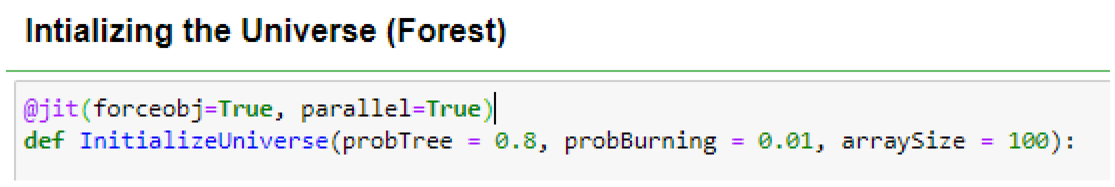


```python
def visualize(grid):

    # Define the color map
    cmap = plt.cm.colors.ListedColormap(['white', 'green', 'red'])
    # Define the bounds for the colors
    bounds = [0, 1, 2, 3]
    # Define the color for each value
    colors = ['white', 'green', 'red']
    # Create a color map object
    norm = plt.cm.colors.BoundaryNorm(bounds, cmap.N)
    # Create a figure and axis object
    fig, ax = plt.subplots()
    # Plot the grid
    ax.imshow(grid, cmap=cmap, norm=norm)
    # Remove the axis ticks and labels
    ax.set_xticks([])
    ax.set_yticks([])
    # Set the title of the plot
    ax.set_title('Spread of Fire')
    # Create a color bar for the plot
    cbar = plt.colorbar(matplotlib.cm.ScalarMappable(norm=norm, cmap=cmap), ax=ax, ticks=bounds, boundaries=bounds)
    #cbar.ax.set_yticklabels(colors)


    def update(frame):
        ax.clear()
        # Update the grid
        ax.imshow(grid, cmap=cmap, norm=norm)
        return ax

    # Create the animation
    anim = animation.FuncAnimation(fig, update, frames=grid, interval=500, blit=True)
    # Show the animation
    plt.show()
```

Methods and implementation: The Architecture of this project is composed of two Python files (namely ACW_Parallel_Implementation and ACW_Sequential_Implementation) in which the code is structured to differentiate Sequential and Parallel implementation as this will give the flexibility to easily measure the runtime execution of each implementation.
The Parallel implementation uses the Moore Neighborhood Algorithm while the Sequential Implementation uses the Voon Neuman Algorithm to determine its transition rule.

The Parallelization of the implementation is done adopting Numba and Pool Multiprocessing; Multiprocessing has some overhead when it is compared to Numba as Numba uses multithreading. The comparison is quite complicated as it sometimes requires tradeoffs when deciding the parallelization method to adopt. The overhead of multiprocessing sometimes makes it the best method especially when complex computation is done on a computer with several processors.

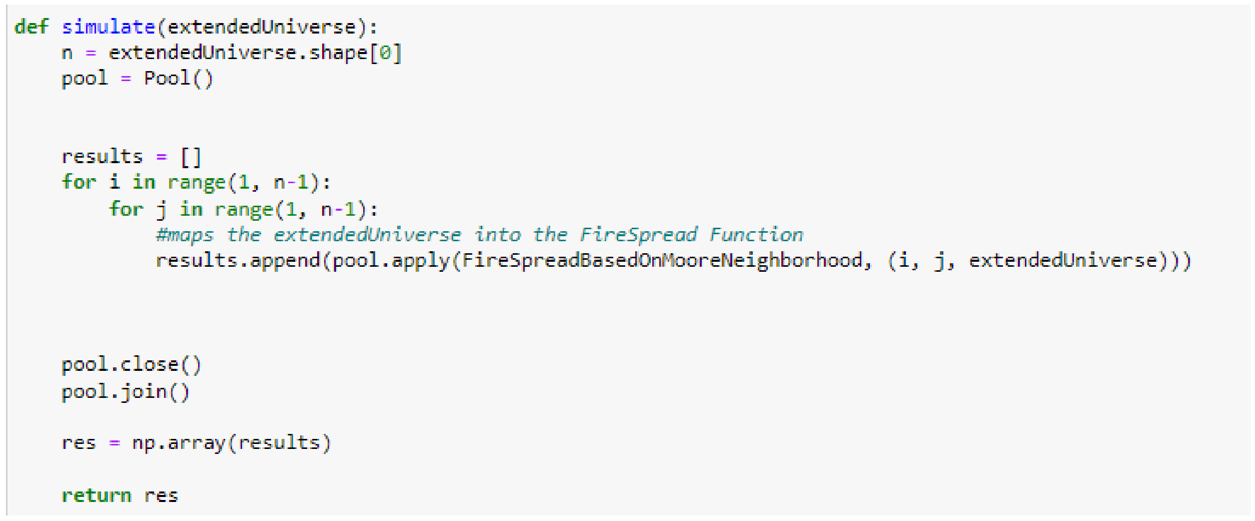
As shown below, The Numba Paralleization technique was used when looping through to initialize or populate the forest grid (universe).
NB: the attribute “nopython=True” wasn’t used due to the deprecated version of Numba.



Intializing the Universe (Forest)

```
@jit(forceobj=True, parallel=True)
def InitializeUniverse(probTree = 0.8, probBurning = 0.01, arraySize = 100):
```

The other part of the implementation where Parallization was applied was on the FireSpread based on Moore Neighborhood Algorithm. This parallelization technique was achieved by applying Multiprocessing against the Fire Spread.

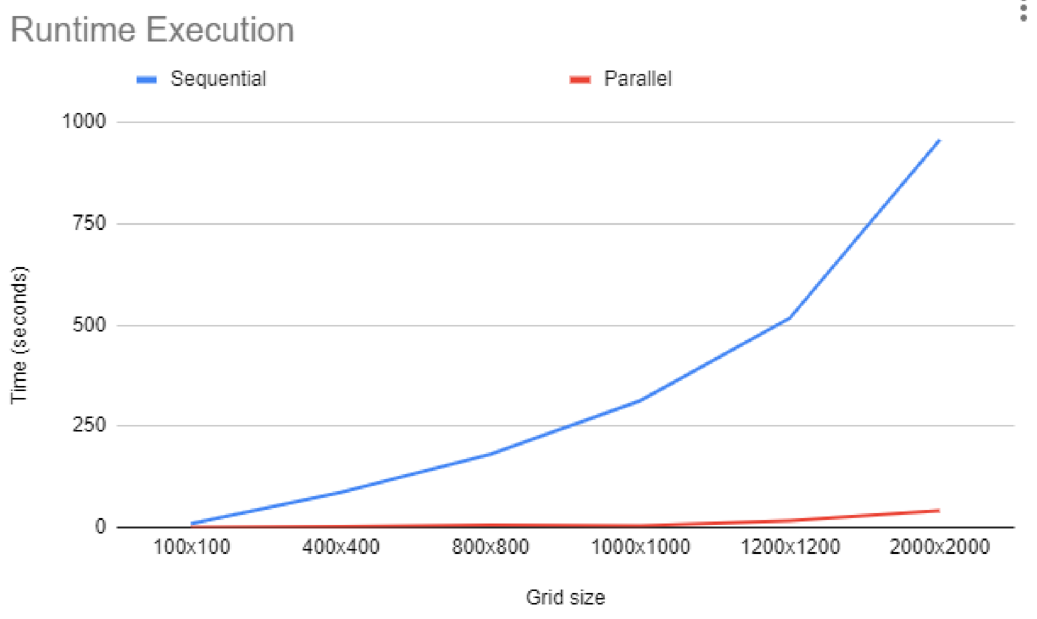


```
def simulate(extendedUniverse):
    n = extendedUniverse.shape[0]
    pool = Pool()


    results = []
    for i in range(1, n-1):
        for j in range(1, n-1):
            #maps the extendedUniverse into the FireSpread Function
            results.append(pool.apply(FireSpreadBasedOnMooreNeighborhood, (i, j, extendedUniverse)))



    pool.close()
    pool.join()

    res = np.array(results)

    return res
```

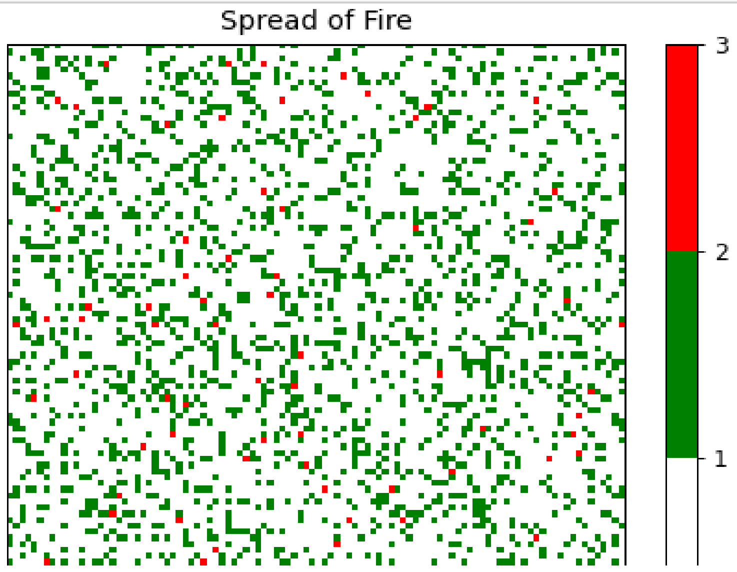
Results and evaluation

Implementation. Grid size	Sequential (Von Neuman Neighborhood)	Parallel (Moore Neighborhood)
100x100	9.147550821304321 => 9.1746 seconds	0.09780573844909668 => 0.0978 seconds
400x400	86.67544567896056 => 86.6745 seconds	1.6115162372589111 => 1.6115 seconds
800x800	180.7939864333278 => 180.7940 seconds	6.697645425796509 => 6.6976 seconds
1000x1000	312.5674335781086 => 312.5674 seconds	10.308103084564209 seconds => 3.7312 seconds
1200x1200	516.7654335754650 => 516.7654 seconds	16.346786737442017 => 16.3468 seconds
2000x2000	956.6781819391061 => 956.6781 seconds	41.86100649833679 => 41.8610 seconds

The Chart below shows the Runtime Execution between the Sequential and Parallel Implementation which was calculated using Python time. This depicts that the Parallelized implementation took lesser time compared to the sequential algorithm. This runtime also varies depending on the Grid Size as larger grid size such as 2000x2000 require higher computation time compared to a 100x100 grid size.



The figure below is a frame from the animated simulation model



Conclusion

The Implementation has been able to showcase the runtime time complexity of both Sequential and Parallel algorithms and how parallelization was achieved using both Numba and Multiprocessing pool to reduce the runtime. Although, this Algorithm still requires some form of refactoring and optimization regarding the structure of the algorithm such that each implemented Neighborhood algorithm can be consumed by any of both parallel and sequential implementations.

References

- Numba. (n.d.). Deprecation — Numba documentation. Retrieved April 15, 2023, from https://numba.readthedocs.io/en/stable/reference/deprecation.html#recommendations
- Numba. (n.d.). njit.parallel=True or python.multiprocessing for speed up? Retrieved April 15, 2023, from https://numba.discourse.group/t/njit-parallel-true-or-python-multiprocessessing-for-speed-up/130/2
- Shiftlet, A.B. and Shiftlet, G.W. (2014). Introduction to Computational Science: Modelling and Simulation for the Sciences. Princeton, N.J.: Princeton University Press, 2nd ed.