Forest Fire Spread Modelling Report

Background

Assuming that an initial forest exist with empty lands, and trees that are not burning and the ones that are burning, this report has been written to describe how cellular models were implemented to spread the fire all over the forest. It will highlight how the forest fire spread has been simulated following influences from burning neighbouring trees and lightning strikes on the site. Evaluations of parallel and sequential implementations of the simulations would be carried out also to discover the performance difference of any of the two processing implementations.

Modelling

Implementing the cellular models needed for this simulation was done in 4 different stages, the first involved creating a two dimensional array that represents the forest grid, and filling it with 0, 1 or 2. The values represent empty lands, trees that are not burning and trees that are burning respectively. And the forest grid was initialized using the Monte Carlo probabilistic approach (Shiflet & Shiflet, 2006) given that the existence of any of the values at any given location on the forest are not certain. Hence the **InitForest** function was created to initialize the forest grid as shown below.

Stage 1

```
3]: def InitForest(rows, cols):
        # initializing the forest with trees, fire, boundary

        forest = np.zeros((rows, cols))

        for i in range(rows):
            for j in range(cols):

                # randomly choosing whether there is a tree c
                # if the random number is less than the probc
                if random() < probTree:
                    # if the random number is less than the p
                    if random() < probBurning:
                        forest[i][j] = BURNING_TREE
                    else:
                        forest[i][j] = NORMAL_TREE
                # else it just an empty ground
                else:
                    forest[i][j] = EMPTY_GROUND

        return forest
```

The function fills the forest grid based on the assumptions

- That a burning tree cannot exist in isolation, hence, the condition that a tree exists in an area was evaluated before evaluating the conditions for burning tree.
- That If the conditions for the existence of burning tree at a location were met, the value of a burning tree would be assigned to the grid location, else a tree is assigned
- That if the condition for the existence of a tree at a location on the forest is not met, then then the empty land value would be assigned to the grid location.

Stage 2

The next stage in the implementation of the simulation model, is the expansion of the boundaries. It entails an addition of an extra layer of cells at the boundaries of the forest to supply temporary neighbours for spreading the fire at the border. The method is known as periodic boundary conditions (Shiflet & Shiflet, 2006).

```
def ExtendSiteBoundary(forest):
    # using periodic boundary conditions
    # extend the site boundary by add one temporary rows to the top and buttom
    # and 2 columns to the left and right boundaries

    # adding the top boundary to the buttom boundary and vice versa
    RowStack = np.row_stack((forest[-1,:], forest, forest[0,:]))

    # adding the left boundary to the right boundary and the right boundary to
    ColumnStack = np.column_stack((RowStack[:,-1], RowStack, RowStack[:,0]))

    # return the expanded forest
    return ColumnStack
```

Stage 3

The next stage in the implementation is spreading the fire. The functions for spreading the fire is displayed below.

```
# converting the von Neumann neighborhood into Moore neighborhood as specified in the assignment
def SpreadTheFireUsingMoore(forest, rows, cols):
    # spreading the fire in the forest
    for i in range(1, rows + 1):
        for j in range(1, cols + 1):
            if forest[i][j] == EMPTY_GROUND:
                # if the area is empty ground, then there would be no fire
                forest[i][j] = EMPTY_GROUND

            # if there is a tree in the area, fire can spread to the area
            elif forest[i][j] == NORMAL_TREE:

                # if a neighbouring area is burning, then the tree burns, we'll check the 8 neighbour
                # north, south, east, west, north-east, north-west, south-east, south-west
                if (forest[i - 1][j] == BURNING_TREE or forest[i + 1][j] == BURNING_TREE or
                    forest[i][j - 1] == BURNING_TREE or forest[i][j + 1] == BURNING_TREE or
                    forest[i - 1][j - 1] == BURNING_TREE or forest[i - 1][j + 1] == BURNING_TREE or
                    forest[i + 1][j - 1] == BURNING_TREE or forest[i + 1][j + 1] == BURNING_TREE):

                    # the tree will not burn it's immune to fire
                    if random() < probImmune:
                        forest[i][j] = NORMAL_TREE
                    # otherwise the it will burn
                    else:
                        forest[i][j] = BURNING_TREE

                # if lightning strikes, then the tree burns
                elif random() < probLightning:
                    # the tree will not burn it's immune to fire
                    if random() < probImmune:
                        forest[i][j] = NORMAL_TREE
                    # otherwise the it will burn
                    else:
                        forest[i][j] = BURNING_TREE

                # otherwise, the tree remains a normal tree
                else:
                    forest[i][j] = NORMAL_TREE
```

It says a tree burns if any of its Moore neighbours is on fire and it's not immune, otherwise, it doesn't does not burn.

It also says if lightning strikes a site, the tree burns if any it is not immune, otherwise, it doesn't burn.

Stage 4:

This stage is all about executing the models and showing and saving the forest animations. The animation was displayed using matplotlib. Please see the zip file uploaded as part of this assignment for the GIF files of the different forest sizes.

Methods of Implementation and Results

The implementation of the parallelization of the models was made possible with the Numba library. And add by adding the jit(nopython=True, parallel=True) to the top of the functions as shown below.

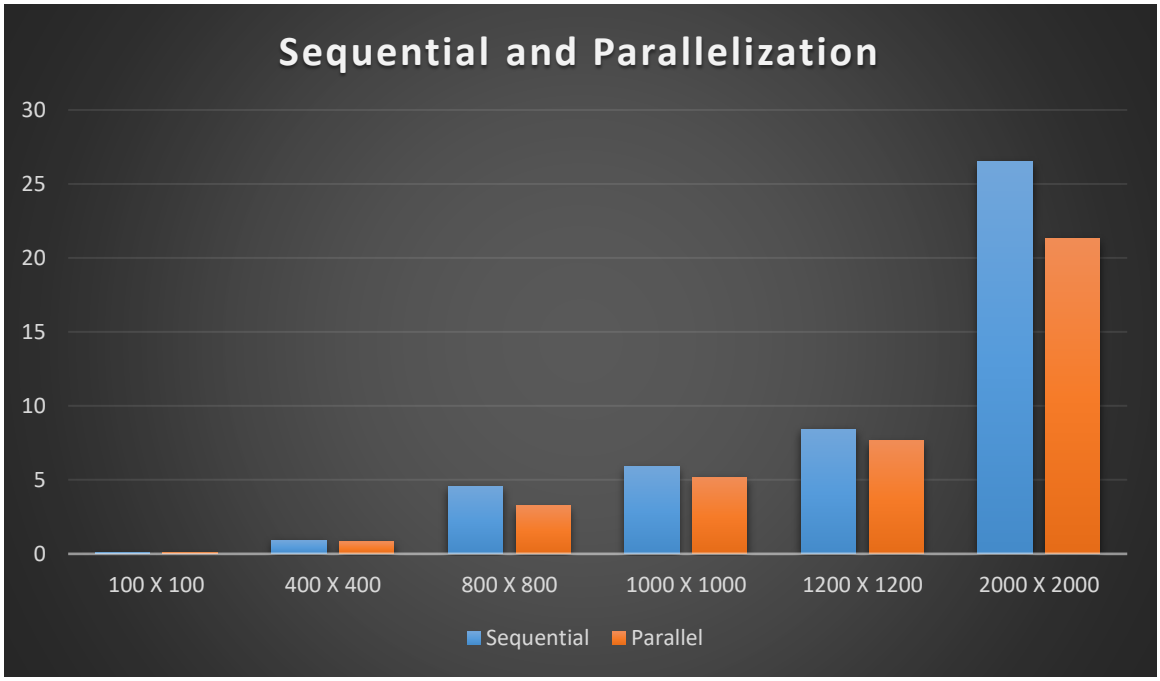
```
jit(nopython=True, parallel=True)
def Parallelized_ExtendSiteBoundary(forest):
    # using periodic boundary conditions
    # extend the site boundary by add one temporary rows to the top and buttom
    # and 2 columns to the left and right boundaries

    # adding the top boundary to the buttom boundary and vice versa
    RowStack = np.row_stack((forest[-1,:], forest, forest[0,:]))

    # adding the left boundary to the right boundary and the right boundary to
    ColumnStack = np.column_stack((RowStack[:,-1], RowStack, RowStack[:,0]))

    # return the expanded forest
    return ColumnStack
```

```
jit(nopython=True, parallel=True)
def Parallelized_InitForest(rows, cols):
    # initializing the forest with trees, fire, boundary

    forest = np.zeros((rows, cols))

    for i in prange(rows):
        for j in prange(cols):

            # randomly choosing whether there is a tree c
            # if the random number is less than the probc
            if random() < probTree:
                # if the random number is less than the p
                if random() < probBurning:
                    forest[i][j] = BURNING_TREE
                else:
                    forest[i][j] = NORMAL_TREE
            # else it just an empty ground
            else:
                forest[i][j] = EMPTY_GROUND

    return forest
```

To help evaluate the sequential and parallel implementation, I recorded the time difference for one iteration of the spread, inclusive of the forest grid initialization and boundary expansion. Here is the table showing the recorded time it took to simulate each of the forest sizes.

Forest Size	Sequential	Parallel

100 X 100	0.063	0.062
400 X 400	0.92	0.82
800 X 800	4.58	3.26
1000 X 1000	5.89	5.17
1200 X 1200	8.42	7.63
2000 X 2000	26.51	21.33



The chart above shows that the parallelized functions executes faster than the sequential implementation.

Conclusion and Future Work

For the future work, even though Numba makes it easier, I will try using the multiprocessing python library as I believe it gives me the leverage to do more with parallelization.

References

Shiflet, A. B. & Shiflet, G. W. (2006) *Introduction to computational science: Modeling and simulation for the sciences*. Princeton; Oxford: Princeton University Press.