Data Distribution

The initial Data Distribution was done by first creating a 2D array of size **[X_limit, Y_limit]** with cells of 0. The 2D array was then passed to the read_input_function which placed 1's in the corresponding coordinate of the 2D array passed, according to the coordinates in the csv file.

With this data, each process then selects the appropriate decomposition of the entire life array. This was done by calculating the amount of rows of the 2D array each processor gets:

```
int rowsPerProcess = X_limit/size;
```

With the amount of rows each processor gets, I then calculated the starting row of each Processor out of the 2D array.

```
int rowStart = rowsPerProcess * rank;
```

I then created a new 2D array with dimensions **[rowsPerProcess] [Y_limit]** for each processor. With this new array, I iterated through the life board to copy the corresponding rows from rowStart -> rowStart + rowsPerProcess.

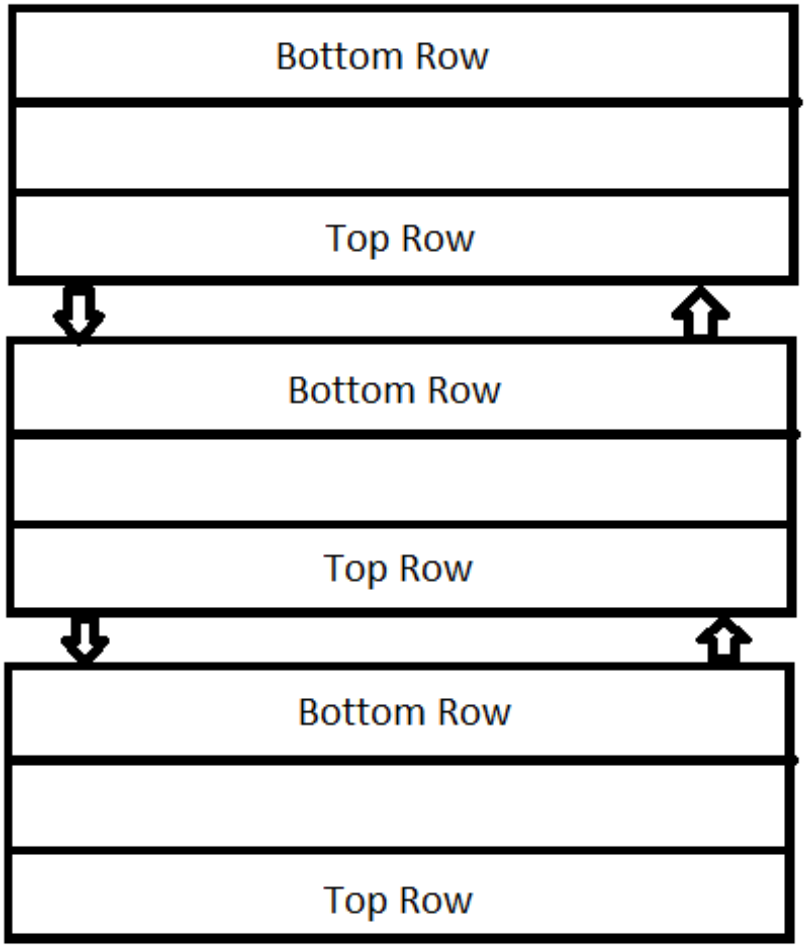
```
int **receivedLife = new int *[rowsPerProcess];
for (int i = 0; i < rowsPerProcess; i++){
    receivedLife[i] = new int[Y_limit];
    for (int j=0; j < Y_limit; j++){
        receivedLife[i][j] = life[rowStart][j];
    }
    rowStart++;
}
```

I then padded the receivedLife array on all sides by 1 and stored it in receivedPrev. These 2 arrays were passed in the compute function instead of the life and previous_life array.

This could've been more efficient by having only the master process reading the csv, creating the entire table, then scattering the corresponding rows to each worker process.

Preventing Deadlocks

During the computational phase of the program, I prevented deadlocks by having every MPI_Send message mapped to the appropriate MPI_Recv. This way, no processes would be waiting for a message to be received before continuing execution.



Using the above diagram as a representation of the message passing algorithm between the processors, each processor would have Send 2 rows and receive 2 Rows (aside from the first and last processor). With this implementation, there would not be a deadlock since each Send message had a corresponding Receive message to unblock execution.

```
if(rank != 0){
  MPI_Send(bottomSendRow, Y_limit, MPI_INT, rank-1, 0,
MPI_COMM_WORLD);
}

if(rank != size-1){
  MPI_Recv(topRecvRow, Y_limit, MPI_INT, rank+1, 0, MPI_COMM_WORLD
MPI_STATUS_IGNORE);
}

if(rank != size-1){
  MPI_Send(topSendRow, Y_limit, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
}

if(rank != 0){
```

```
  MPI_Recv(bottomRecvRow, Y_limit, MPI_INT, rank-1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
}
```

With the way my algorithm was structured process rank = 0 and process rank = size - 1, would receive no bottomRow and no topRow respectively since they have no rows above and below to check. In order to stop deadlocks, I did not have send or recv messages for the bottomRow of rank = 0 and topRow of rank = size - 1. Instead, I just instantiated a new 1D array, with 0's, of size Y_limit.

Performance

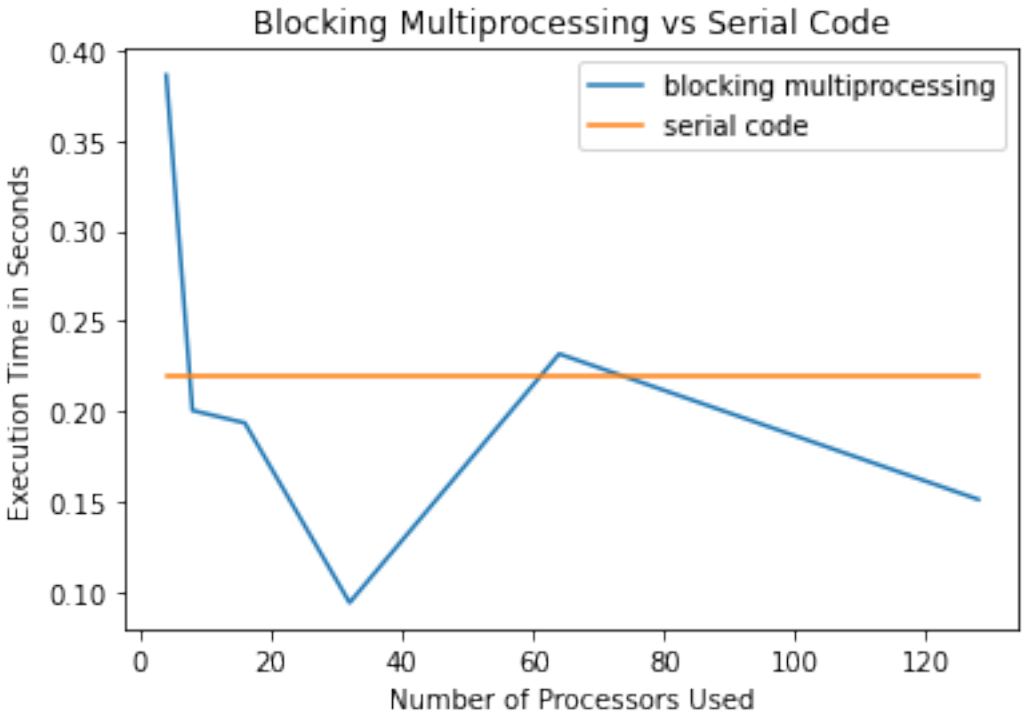
```
import matplotlib.pyplot as plt

cores = [4, 8, 16, 32, 64, 128]
serial = [0.220122, 0.220122, 0.220122, 0.220122, 0.220122, 0.220122]
blocking = [0.386293, 0.20064, 0.193721, 0.094214, 0.231931, 0.151348]

plt.plot(cores, blocking, label = "blocking multiprocessing")
plt.plot(cores, serial, label = "serial code")

plt.xlabel('Number of Processors Used')
plt.ylabel('Execution Time in Seconds')
plt.title('Blocking Multiprocessing vs Serial Code')
plt.legend()

plt.show()
```



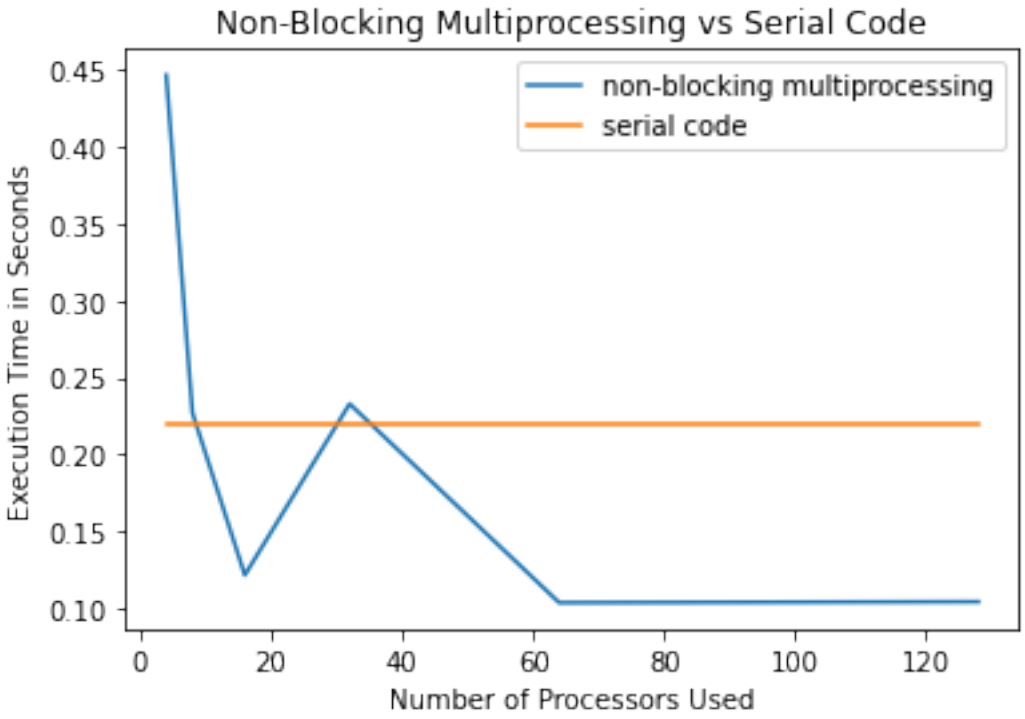
```
import matplotlib.pyplot as plt

cores = [4, 8, 16, 32, 64, 128]
serial = [0.220122, 0.220122, 0.220122, 0.220122, 0.220122, 0.220122]
non_blocking = [0.446384, 0.227174, 0.121757,  0.232913, 0.103862,
0.104467]

plt.plot(cores, non_blocking, label = "non-blocking multiprocessing")
plt.plot(cores, serial, label = "serial code")

plt.xlabel('Number of Processors Used')
plt.ylabel('Execution Time in Seconds')
plt.title('Non-Blocking Multiprocessing vs Serial Code')
plt.legend()

plt.show()
```



Performance Results

The results of the performance test were not exactly what I expected. The simulation with the serial code was faster for both the blocking and non-blocking simulations with 4-cores. Although there is overhead for initializing MPI and passing data between the cores, I still expected the performance for 4-cores to have a speedup of around 4, compared to the serial execution. The same goes for 8-cores, where the times for serial, blocking and non-blocking are almost equal.

When we get to 16-cores, the blocking and non-blocking multiprocessing simulations are both faster than serial execution, and non-blocking is much faster than the blocking implementation. This was expected behavior.

With the 32-core simulation however, the non-blocking jumps in execution time compared to the blocking. This was unexpected since it should theoretically still be faster than the blocking code. Even though I ran the simulation for 32-cores multiple times, this behavior persisted. I'm not sure if this was due to my implementation or Zaratan having variability in execution times. Both were still faster than the serial code

During the 64-128 implementation, the non-blocking code remained consistent in times, while the blocking had spikes in execution times. This is slightly expected since more processors adds more overhead and communication between processes, while the asynchronous code should not be affected as much since it isn't blocking.