Umut Deniz Şener – 2018400225
Leyla Yayladere – 2018400216

MPI PROJECT REPORT

(a) INTRODUCTION:

In this project, we are asked to implement a parallel algorithm for simulating a game using an MPI library, so that we would gain some experience on parallel programming.

We completed the project successfully and we got the correct result for each input case provided. After making few corrections, we didn't encounter with any deadlock.

Our code compiles and runs successfully with the command "*mpiexec -n [P] python3 simulator.py input.txt output.txt"*.

(b) STRUCTURE OF THE IMPLEMENTATION:

Reading from input files, writing to output files, and distribution of data to worker processes are done by manager process. We keep the data as 3D array which contains a location map for towers and their corresponding health map.

We assume that N is always divisible by number of processors minus 1 and we applied striped approach for splitting the data. Therefore, communication among processors is needed only for taking upper and bottom-line information.

Communication Part:

Firstly, manager process reads from input file and creates location and health map accordingly. Then, it sends the corresponding parts/cells of updated location and health maps to each worker processes.

To prevent deadlocks, we separated worker processes as even and odd accordingly to their ranks. Firstly, in each worker process, the group of N (size of map) / p – 1 (number of worker processes) adjacent rows of both location and health maps of the towers. Then for even ranked worker processes, first row in the corresponding cells of location map is sent upper neighbor and last row in the corresponding cells of location map is sent bottom neighbor. Of course, we considered the edge cases and didn't send the data if process has no upper/bottom neighbor. These sent data is received by odd ranked worker processes, in this way we prevent deadlocks. With the same logic, odd ranked worker processes send the required data to even ranked worker processes.

After eight rounds, in other words after each wave, both updated location and health maps is sent to manager process by each worker processes. Then, worker process adds new towers for next

wave and sends the corresponding parts/cells of updated location and health maps to each worker processes. Then game and communications are continued in same way for each wave until the end of the last wave.

Game Logic Part:

Game simulation is basically done by the help of few functions. We'll explain them briefly.

DifferentLineAttack_o(Attacker, Defender) → arranges the attacks of the "o" towers to "+" towers in upper/bottom rows according to their attack power and pattern, then returns an array that contains the damages in each column corresponding defender row

Example: Attacker = ["o", ".", ".", "+"] &Defender = [".", "+", ","o"] → Damages = [0, -1, 0, 0]

DifferentLineAttack_x(Attacker, Defender) → arranges the attacks of the "+" towers to "o" towers in upper/bottom rows according to their attack power and pattern, then returns an array that contains the damages in each column corresponding defender row

Example: Attacker = ["o", ".", ".", "+"] &Defender = [".", "+", ","o"] → Damages = [0, 0, 0, -2]

SameLineAttack_o(Attacker) → arranges the attacks of the "o" towers to "+" towers in same row according to their attack power and pattern, then returns an array that contains the damages in each column corresponding defender row

Example: Attacker = ["+", "o", "+", "o"] → Damages = [-1, 0, -1, 0]

SameLineAttack_x(Attacker) → arranges the attacks of the "+" towers to "o" towers in same row according to their attack power and pattern, then returns an array that contains the damages in each column corresponding defender row

Example: Attacker = ["+", "o", "+", "o"] → Damages = [0, -4, 0, -2]

HealthUpdater(CurrentHealths, Damages) → updates the health map of towers according to damage taken

Exmple: CurrentHealths = [5, 1, 2, 3] &Damages = [0, -1, -3, 0]→ UpdatedHealths = [5, 0, -1, 3]

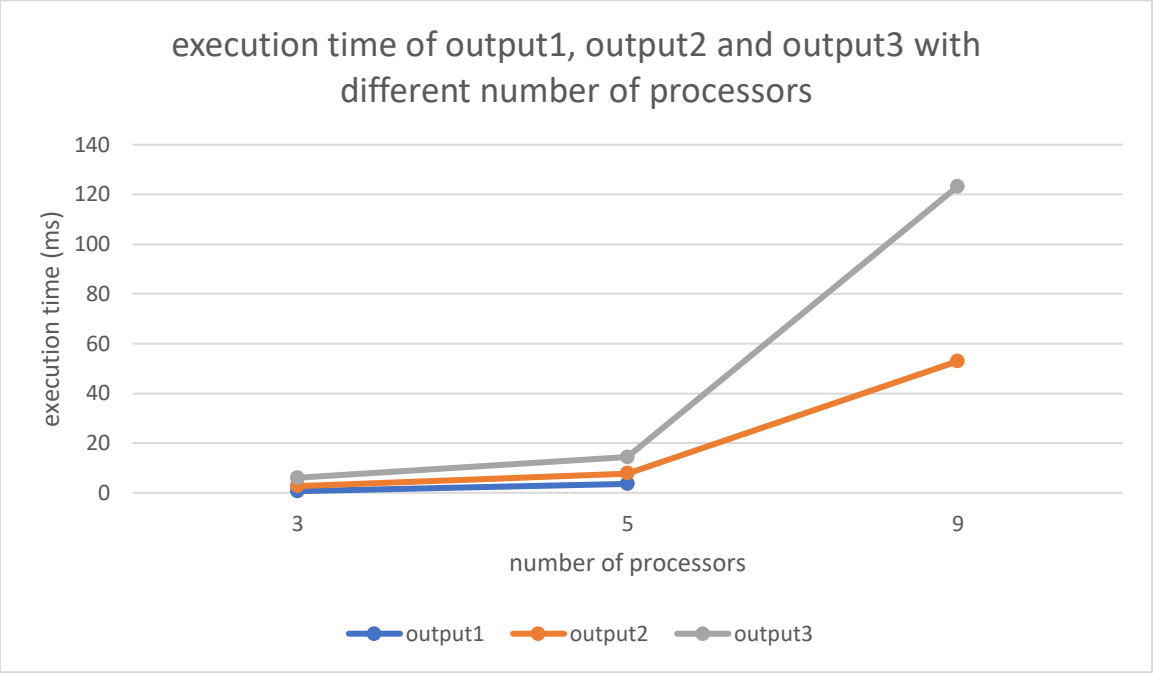
Destroyer(CurrentData, CurrentHealths) → destroys the towers if their health smaller than or equal to zero

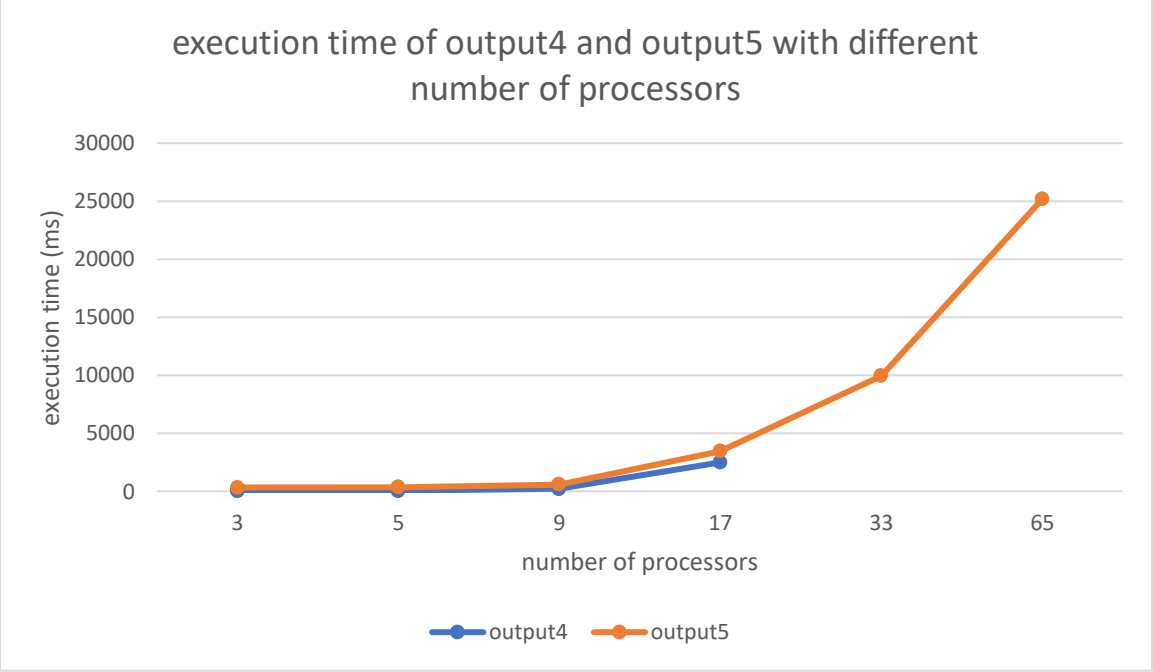
Exmple:CurrentData = ["o", ".", "+", "o"] & CurrentHealths = [5, 0, -1, 3] → CurrentData(which is returned) = ["o", ".", ".", "o"]

Umut Deniz Şener – 2018400225
Leyla Yayladere – 2018400216

In each worker process, we used these functions accordingly after taking the upper and bottom-line location data from neighbor worker processes.

(c) ANALYSIS OF THE IMPLEMENTATION:





Umut Deniz Şener – 2018400225
Leyla Yayladere – 2018400216

As can be seen in the charts parallel implementation with higher number of processors tent to show less performance due to MPI communication overheads among processors. This shows that our inputs are not big enough to run better with higher number of processors. Therefore, amount of running processes should be limited according to the data set size for better performance in real life.

Our other observation from these charts is the fact that communication operations among processors is the bottleneck in terms of execution time. Therefore, we will consider communication operations as basic operations. There are three different types of communications.

First one is sending information from manager process to all other worker processes, this is done in the beginning of each wave. Since manager process sending information to workers one by one, that is why this is done sequentially.

Secondly, every worker process is sending **health and location information** to manager process at the end of each wave and manager process is receiving these data again in sequentially manner.

Considering complexity of these two communication types we can say that **in each wave** there are **2*(P-1)** receiving operations and **2*(P-1)** sending operations as basic operations in manager process where P is total process number. Since other receiving and sending operations are done in parallel manner in worker processes, we don't need to consider them.

Lastly, communication between worker processes is done in every round. Each worker process needs the information of its upper neighbor line and bottom neighbor line except the edge cases. The information consists of **two lists** which show location data of towers. Since there are **8 rounds** and all these workers runs in parallel manner, we can say that **in each wave** there are **2*8** receiving operations and **2*8** sending operations as basic operations.

We calculated above how many lists will be sent or received but we need to also mention that since these lists are the rows of n sized maps. Their length will be $\sqrt[2]{n}$ and actually that much information will be sent or received. Therefore, we need to multiply the result found with $\sqrt[2]{n}$.

To sum up, we have **[2*(P-1) + 2*(P-1) + 2*8 + 2*8] * $\sqrt[2]{\mathbf{n}}$ * W** many basic operations where W is total number of waves. Since W is a constant variable, time complexity is **O(P * $\sqrt[2]{\mathbf{n}}$)**.

**note: we assumed that our basic operations are sending and receiving information, but the result will not change if we assume basic operation is only sending or receiving. Because number of these operations are equal (O(P * $\sqrt[2]{n}$) = O(2 * P * $\sqrt[2]{n}$)).

Umut Deniz Şener – 2018400225
Leyla Yayladere – 2018400216

(d) TEST OUTPUTS:

Output1:

```
o . o .
. . . .
. . + +
o . . +
```

Output2:

```
+ + . o . . + +
+ . . o . . . +
+ + . o . . + .
+ . . . . . . .
+ . . . . o o .
+ . o . . . . .
. . . o . . . o
+ + . o o . . o
```

Output3:

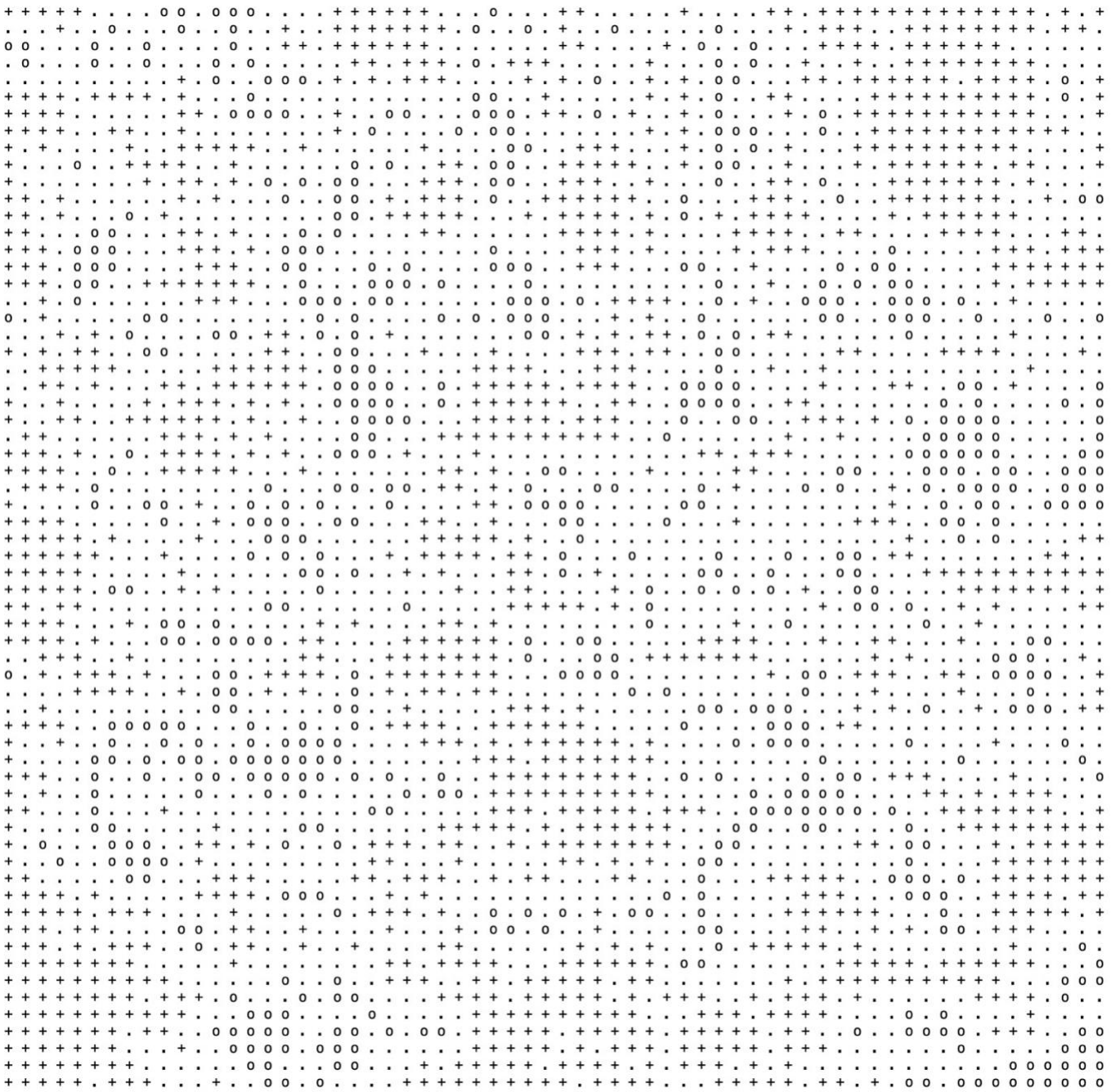
```
+ + + + . + + +
+ + + . . . + +
+ + . + . . + +
+ + + + . . . +
+ + + + . . . +
+ + + . . + . +
+ + + + + + + +
+ + + + + + + +
```

Output4:

```
+ + + . . + + + + + + + + + + +
+ . . + + . . . + + + + . + + +
+ + . . . + . . + + + + + . + +
+ + + . + . . . + . + . + + + +
+ + + + + + + . + + + . + + . .
+ + + + + + + + + + + + . + . .
+ + + + + + + + + + + + . + + . o
+ + + + + + + + . . + + + + + . .
+ + + + . + + . + + . . . . + .
+ + . . . + . . + + + + + + + .
+ . . . . . . . . . + + + + . . .
+ . . o . + . + . . + + + + + + +
+ . . . . . + + . + . . + + + + +
+ + . . + . + . . . . . . + + +
+ . . . + + + . o . . + + + + +
+ . . + + + + . o o . + + + + +
```

Umut Deniz Şener – 2018400225
Leyla Yayladere – 2018400216

Output5:



Umut Deniz Şener – 2018400225
Leyla Yayladere – 2018400216

(e) DIFFICULTIES ENCOUNTERED AND CONCLUSION:

We got many out-of-bounds errors for our data arrays. To eliminate these errors, we just initialized all data arrays as '.', all health arrays as '0' and return the functions as empty array if the attack is not applicable i.e edge case. This makes a lot easier to calculate edge cases. For example, if a row doesn't have any upper row then we accepted this line as an empty one and applied the game rules.

We encountered one strange deadlock and we couldn't understand its occurrence reason for a while. We were sure that our communication among processes is completely correct and also, we got the correct output even when this deadlock occurred. Simply, our code could give the output but not terminate. Then, we realized that we closed the output file in the end of main code but it should be closed in manager process after writing the output. It was a silly error but it made us to think about all communication part and the possible reasons of deadlocks ☺

In this project, we gained a lot of valuable insight and experience in parallel programming and developing and executing a MPI Project. Also by running the simulation with different configurations (inputs and processor numbers), we saw that in parallel programming, communications between processors creates a bottleneck for execution time and the execution time increases as because of that. That's why parallel algorithms generally take communications between processors as basic operation under analysis.