Hasan Baki Küçükçakıroğlu **2018400141**
Miraç Batuhan Malazgirt **2018400156**

a) Introduction:

Yes, we completed the striped version of the project. It runs successfully for all inputs and worker numbers. We used Python to implement the project. “mpiexec -n [P] python game.py input.txt output.txt “command can be used to execute a project from the terminal.

b) Structure of the Implementation:

Our code builds an environment with a given number of processors as arguments. The first of these (rank=0) is the main processor, this processor that distributes data to all other processors. The main processor takes the data from the input file at the beginning of each wave, processes it and sends it to the worker processors. Worker processors, on the other hand, receive this data and perform the necessary communications. It simulates the rounds and sends the output to the main processor when all the waves are finished. The main processor prints the incoming final output into the output file given as an argument. We will now examine all these processes in detail.

The main processor reads two lines for each wave from the output file. The first of these lines shows the positions of the 'o' towers to be placed, and the second shows the positions of the 'p' towers. Let's say we have p worker processors. Since the dimensions of our game board are N*N, each processor has N/p lines. The main processor sends each worker a list of N/p lists, each containing two maps. The lists in this list contain the position information of the towers of the rows that the processor is responsible for. For example, if each processor is responsible for two rows of lines, a list such as [[{}, {}],[{},{}]] is sent to the first processor. The first list in this list holds the data of the first row for which the processor is responsible, and the second list holds the data of the second row. Thanks to this design, the need to save the line numbers while keeping the position information of the towers is eliminated. The two maps inside the row lists hold the column indexes of the 'o' and 'p' towers, respectively. Column indexes map the life of the tower in that index. An example data list that is generated on main processor,

[[[{0: 6}, {}], [{2: 6}, {}]], [[{0: 6}, {1: 8}], [{}, {}]], [[{}, {1: 8}], [{}, {3: 8}]], [[{0: 6}, {}], [{}, {3: 8}]]] (There are 4 worker processors, each responsible for two rows.)

After each data is parsed and sent to the relevant processors. Each worker takes this data and keeps it in a similar list. For the first data (first wave), this process is just copying the incoming list to the local list. In waves, each processor compares the incoming data with its local data. If the corresponding column of the related row is empty, it adds the tower that needs to be added there. If this part is full, it does not add the tower and continues to do this for other towers.

Now we talked about how the homeland reaches the worker processors for each wave (first and the others) and how it is processed and converted into local processor data here. In the next process, the rounds start. Each wave is 8 rounds. The necessary parts of the data, which reaches the processor at the beginning of the wave and is processed into local data, are sent to the neighboring processors before the fight begins. The reason for this is that while each processor has only its own data, it cannot control whether the towers located on the borders of the neighboring processors attack their own towers. To check this, the border lines of the neighboring processors (for the striped version, only the lower and upper processors) should be sent to the border neighbors. Each processor has two lists, named neighbor_lower and neighbor_upper, to hold two rows of information from its upstream and downstream neighbors, respectively. After these lists are filled with data from neighbors, nothing remains but war operations. Now I will explain how this communication took place, the decisions we made during this communication, the points we paid attention to and the design.

It is guaranteed that the number of worker processors will be even. Therefore, the rank of the highest processor will always be odd and the rank of the lowest processor will always be even. All these processors cannot send their upper and lower lines to each other at the same time. In fact, not every processor can send only the bottom line or only the top line at the same time. The reason for this is the possibility of deadlock formation. Even if each processor sends its bottom line at the same time, send operations will create a deadlock because no one recites the sent data. Therefore, every send operation must have a corresponding recv operation. To achieve this, we designed the odd,, even processor as we discussed in the lecture. According to this design, the processors with odd rank will send the lower limit lines to the processors with even rank below them, and these processors will receive these lines. Then (except for the first row) single rank processors will send the upper limit rows to the processors with even rank above them, and even processors will receive this data. Then the sending order will pass to the processors with even rank. The processors with even rank will send the upper limit lines to the processors with an odd rank above them, and the processors with an odd rank will receive them. Then the even rank processors (except the lowest) will send the lower bound lines to the odd rank processors. Single processors will receive these lines. When all these operations are successful, each processor will have 2 more adjacent rows next to its N/p row. Thus, all war operations will be able to be done without a setback.Now let's explain how the war operations take place.

Calculates how much damage each of the towers in each processor list takes. To do this, it is checked for each tower in the list to see if there is a tower around it. For 'o' towers, only the right, left, bottom and top positions are checked to see if there is an 'o' tower; For '+' towers, look for an 'o' tower for each of the 8 surrounding positions. The number of enemy towers around the towers is recorded and the damage to the towers is calculated by calculating the towers' health. If the tower's health drops to zero or below, that tower's health is reduced, but the tower is not removed from the list. Because if we removed this tower from the list, we would be ignoring the damage this tower could cause to its unchecked neighbors. For this reason, towers that need to be removed are kept in a list and only removed at the end of each round. At the end of every 8 rounds passed in this way, a wave is completed. If a new wave is to start, new data is received and all operations are repeated. If all waves are completed, the final local processor data is processed and sent to the main processor to be written to the output file.

The main processor, which reaches the final data (by recv), takes the tower indexes here and prints them to the output file. For non-tower indexes are written '.' Let's talk a little bit about why we chose the data structures we use. Instead of sending rows completely from the main processor to the worker processors, we avoided iterating through the whole row to first search for o and p towers by using a map for the 'o' and 'p' towers, where the column indexes took their toll. In the search for an enemy tower, we only checked whether the required column key exists in the relevant map. Keeping the health of each tower on the map prevented using a separate structure for the health processes of the towers. Each tower is accessible and editable with O(1) complexity. When searching for enemies on the right and left borders, it was sufficient to check if a column key outside the index exists in the map (it will never be found). In this way, we did not have to consider these cases separately.

c) Analysis of the Implementation:

Since send-receive operations are the most time taking operations - processor communication takes most of the time - we are taking them as a basic operation and we carry out our analysis.

Since every send - receive pair occurs parallely they are counted as a basic operation. Here is our communication scheme which explains our send - receive pairs.

In every wave the main processor sends its data to the all P processor. Therefore there is a P send-receive pair. (Since they are consecutive). In every 8 rounds of each wave the following operations are executed.

First of all, odd numbered processors send their row information to the even numbered processors. Even numbered processes receive these rows. Since processor count is an even number all odd numbered processors can do this sending operation. Therefore P/2 send and

receive pairs can do their operations parallely. Since these operations are done parallely, these operations are basic operations.

Secondly, odd numbered processors send their row information to the even numbered processors. Even numbered processors receive this data. Since the first processor cannot send the data to the upwards, this processor cannot do this operation. Therefore (P/2)-1 send-receive pairs can do their operations parallely. Since these operations are done parallely, these operations are basic operations.

Thirdly, the even numbered processors send their low boundary rows to the downwards odd numbered processors. Odd numbered processors receive these rows. Since the last processor cannot send its data to the downwards odd numbered processor, this last processor cannot do this operation. Therefore we can do (P/2)-1 send-receive pair parallely. Since these operations are done parallely, these operations are basic operations.

Lastly, all processors send their data to the main processor at the end of the all waves. Therefore, P send receive pair can do their operations parallely. Since these operations are done parallely, these operations are basic operations.

Regardless of the case(worst, average, best), the following number of basic operations are executed.

> Exactly W*[P+ 8*4]+P communication operation will be executed.
> Exact number of basic operations : W*P+32*P+P= W*P+33*P

Best Case: O(W*P)
Average Case: O(W*P)
Worst Case: O(W*P)

Our answer of what factors speed up/down our implementation is the following. Since we use ‘map’ to send our coordinates we can decrease the time complexity of search operations from O(n) to O(1). However in order not to be subject to deadlocks, we carry out send-receive operations in four phases. This approach speeds down our implementation. Also another factor that speeds down our implementation is that we send the data in the beginning of each round rather than sending all the data at once. This causes extra send-receive operations and speeds down the implementation.

d)Test Outputs

test_input1.txt's result:

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

test_input2.txt's result:

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

test_input3.txt's result:

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

+.++..+...++++++
+++++.....++.+.+

test_input4.txt's result:

o o o . o . . . + + . + . . . o . . . . o . . . . o . . . + . + . . o o . + . . o o o o o . + + + + ... + . o .
o . .
. . o o . o . . + + + + + . . . o . . o . . . o . . . . + + + + . . o o . . . o o . . o . . . . o o . + + . + . o .
. o o
. . . o . . . . . + + . + . . . o o o . . . . o . . . + + . o o o . . o . . o o . o . . . + . o o . . . . . + . o o . .
.
. . . o . . + . . + . + . . + . o . . . . o . . o o o . . + . . . o . o . . . o . o . . o . o . . . . . + ...
. . . . + + + . . o . o . . . . o o . o . + ... o . . . . o . o . + ... + ... + . . +
+ . + + ... + . + ... + ... o . . . . o . . o . . . . . + + . . o o . . . . o . + + . o . o o . . . . .
. . . . . + . . o . . + + + . + ... o o o . . . . o . . . . . o o . . . + . o . . . . + ... o . . . + +
+ . o . . . . . o o . . + + + . + . o . o . . . . o . o . . . . . + . o . . . . o o . . . . . o . . + . o . . . . + +
+ . . . o . + ... + . + + . + . . o o . . . o . . . . + + + + + ... o o . . . . o o . . . +
+
+ . . . o . . . . + + + ... o o o . o . . o o . . . . + + + + . . + ... + . . . o o . . . . .
+ + . o o . . . . . o o . + + + + + . + + ... o . o . . o . . o o . . . + + + ... o . . + . . . o . . o . + + .
+ . . . . . . o o o o . + . + ... + + . . . o o o o o o . . . . . + . + + + . o o o . o o o . . . . . o o o . . . + + +
. . o . . . o . o o o o . . + ... o . o o o o o . . . + . . . + + ... o o . o o o o o o o . . o . . + + +
o o . . . o . o . . . o o . . . . + . . . o . . . . . . o . . . . o . . . . . o o o o o o . . . o . + + + +
. . . . o o . . o . o . o . . . . . . o o . . . . o . . . . o . . . . . . o . o o o o o . . . . . . + . +
. + + . o o o o . . o . o . . . . o o o . . . + . + + . o . . . . . . o o o o . . . . . + . o o . o o . . . o . . . o o .
. + +
+ + + . . o o o . . . o o o . + . . . o o . . . . + ... o . . . . . o . . . o . o . . . . o o o o o . . . o . . . . o .
+ +
. . + ... o . o . . . . . o o o . . . . . o . . . . + . o o o . + . o . . . . o . o o . . o . + + ... +
. . . . . + . + . o o o o . . . . . o . o . o o . . o o . . o . . . . o . . . . o o . + . + . + . o . . . o . o . . . . o .
+ +
. o . . + ... o o . . . o o o . . o o o o o . . . o o . . o . . . . o . . . . . + ... o . + . . o . + +
. . . . o o o . . . . . . o . . . o o o o . o o . . . o o . o o o . . . . + + . . . + . + + ... + ...
+
. . . o o o . . . . o o . . . . . . o o . . o o o . o . . . . . o . . o . . . . + . o . . . . + + + . o o . . + + . . +
+
o . o o o o o o . o o . . o . . . o o o o o o o o o o o o o . . o . . . . . + ... o o . . . . . . o . . . . +
+ . + +
. . . o o o . . . . o o o o o . o . . o o . . o o o . . . . + . . o . + . + . o o o . . o o . . o o o o . + .
+ + +

. . . . . + ... o . o o o o o o o o o . o . o o . . . . . + ... o . . . . . o . o . . o . o . . + + +
+
. o . o . o . . . . o o o . o o o o o o o o o . o o . o . . . . . o . . o o . . . o o . . . . . o o . . + . .
+ +
. o o o o o o o o . . o . o o o . . . . o . o o o o o . . + . + . . o . . o . . o o o o . o . o . . + . . . o o . o .
. . . . +
. o o . o o o . . o . o o o o o o . o . . o o o o o . . . . . . o . . . . o o . o o . o . + . o o o . . . o . .
. + +
o . o o o o o o . o o o . . o . o . o o o o o o o o . + + . . o o . . . . . . + . o o o o . . . . + . . o . o . . .
o . . . .
. . o o o o o o . o o . . . . . . o . o o o o o o . . . . o . o o . . . . o . . . o o o o . . . . + ... o . . o o .
. o
+ . . . o o o o o . . o . . . o . o o o o . . . o o o . . o . . o o . + . o o . o o . . o o . . + . . . + ... o o
o o o o
. . . . . o . . o . . . . . o o . o o o . . . . . o o o o o . . . . + . + . . + + + + . . o o o o .
.
. . . o o o . . . o o o o o . o o . . o o . + . o o o o . . . . . o o . . + + . + . . o . o . . . . . + + . + + . . . o
o o . o
+ + . . . o o . o o o o o o . o . . . . . + . . . o . o . . o o . . . . + ... + . + . + + + + + . . . o
o o .
+ + . + . . . o o o . . o . o . . . + + + . . . o o o . . . o o . . o . o o . . . . . + . . . + . + + + ...
o .
+ + . . . o o o o . . o . o o o o . . + + . + + + . o o o o o . . o . . . . o o o . + + ... o o . .
o o . .
+ . + . . o o o o o . . . o o . . o . . . . o o . o o . . . . . . o . o . . . + ... o . . o . o . . o . . . .
+ + . . o o o o . . + . . . o . . o o . . . o . . . o o o o . . . . . . o o . . . . . + . o . o . . o o o . o o o . . .
. o
. + ... o o o . . . . . o . o o . o . . . . . o . o . . . . o . . . . o . . . . + + + . o o . o o o o . . o o o o o
o . .
. . . o o o o o . . + . . . o . o . . . . . o . . . . o . . . . o . . . o . o . o o . o . o o .
. . o . o o o . o . . . o o o o o o . + . . o o o o o o . o o . . . . o . . . . o . . o . . . .
o
+ . o . . o . o o o . . o . o o o o . . . . . o o o o . . . o o o o o . . . o . . . . o o . o . . . . o o . .
. o
. . o . . o o . . . o o o o o o o . . . . . o . . . . . + . . o o . o . . + ... o o . o o . . o . + + + . . o . .
. o
o o o o . o o o o . o o . . o o o . o . + . o . . o . . + . . . o o o . + + . + . . o . . . . o . . . . o . . + . . . o
o . o o
o o o . . . o o o o o . . . o o . o . . . . o . o o . + . + . . o o o . . . . + ... o . . . . o . . . + . . . o o o
o o
. . . . o . o . . o . . + . . . o o o . . . . o o o . . . . . o o . o . o . . . . + . . o . . . . . o . . . . . o . . o o o o
+ + ... o . o . . . o o . . o . . . . o . o o . o o . . . o o . o o . . . + ... o . . . + ... o o o o o
o

e) Difficulties Encountered and Conclusion

We were first getting half correct outputs in the first version of our code. We tried hard to solve the problem. After a day of hard work we figured out that we were deleting the towers in the middle of the round but it should have been at the end of the round.

Also the probability of encountering a deadlock formation has gave us hard times. We took our pencils and A4 sheets and tried to figure out what was going on processors. This experience gave us a practical view of what we have learned in CMPE 322.

In all of the courses there has been a touch of parallelism topic but now all stones fit their places with this project and thread project we did in CMPE322.