CMPE 300 - Analysis of Algorithms
Project 2 Report

Berkay Buğra Gök
2021400258

Talha Özdoğan
2021400246

December 23, 2024

Contents

1 Introduction

This is the report for the second assignment of the CMPE300 class. The goal is to simulate a battle environment among factions on a grid using message passing interface and utilizing multiple processors.

We are given a grid description and we are asked to simulate the waves and rounds for the interactions of the units. There are 4 different types of factions and each faction has different stats. Throughout the simulation, the units from these factions attack, take damage, heal and sometimes move. Our task is to simulate these waves and rounds, and return the final state of the given grid.

2 Design Decisions and Assumptions

We preferred checkered partitioning for the project. The biggest challenge of the checkered partitioning approach was the fact that there were a lot of cells which might be modified by different processors. Especially, the air unit was the source of this issue. Since it might move one cell and might attack targets which were two cells away, the information of the three cells away from the grid of the processor was required. To overcome this issue, we decided to extend the grid of each processor by three extra rows and columns. Another approach would have been using requesting the information of all cells within 3 cell distance at every round from the neighboring cells. However, this would have resulted in too much communication overhead. Instead, with our approach, keeping the track of the these cells became easier. However, we still needed to keep these cells synchronized. To do this, we divided every processor's field into three main regions, which were called Region1, Region2, Region3.

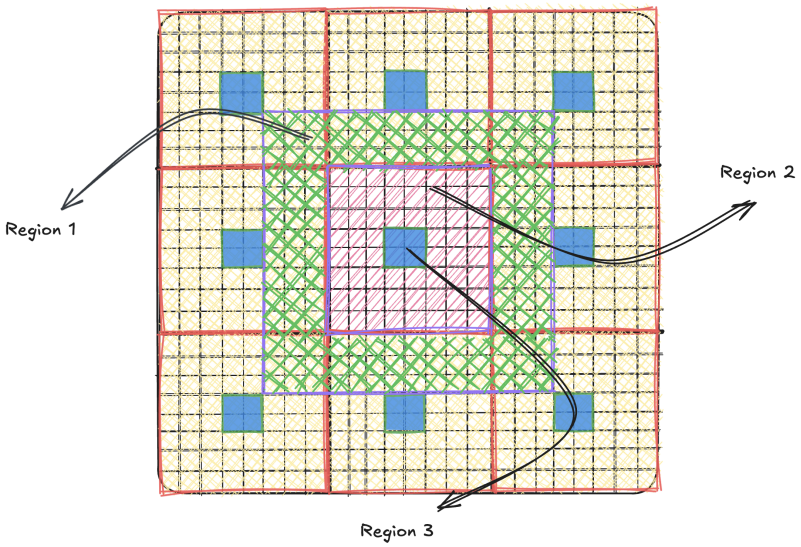

Figure 1: Visualization of each region

Region3 is the core portion of each processor's field. The neighboring processors do not need the information of this region. This is simply because it includes only cells

with more than 3 cells away from neighboring processors' edge cells.

Region 2 comprises the remaining part of the grid that the processor is responsible for. Together with Region3, these regions constitute a processor's grid. However, unlike Region3, Region2's cell information is also important for neighboring processors since Region2 includes cells which are at maximum three cells away from the neighboring processors' edge cells, . Thus, the moves and actions occurred in these cells are sent to neighboring cells as packages.

On the other hand, Region1 is the region that is kept in the processor, but is not in the responsibility region of the processor. It consists of Region2's of neighboring processors and only used for making decisions for the units in Region2. The actions and moves of the units in this region are determined by the neighboring processors since these cells are in the Region2 of these processors. The synchronization of these units are achieved by packages. To keep the Region1's synchronized, at the end of the each windrush, action and flood phase, all processors exchange the information of the moves/actions that took place in their Region2 with their neighbors. Since Region3 of each processor does not intersect with any other processor's Region1, information of the moves/actions in Region1 are not exchanged with neighbors.

In terms of assumptions we have made, we have assumed that the number of worker processors will be a perfect square. With this assumption, we have created an $N \times N$ grid with $N^2 + 1$ processors, one of which is master and others are workers. Also, we have assumed that the edge length of the board is divisible by the $\sqrt{\#worker\ processors}$. This assumption allowed us to divide the board into processors evenly.

3 Implementation Details

3.1 Classes

Before starting the project, we brain stormed about how we can design our implementation. After some time, we decided that we want to have 2 main types of objects which are Workers and Units. We used a base Unit class which we extended each type of units from based on their factions. So, besides the base Unit class, we have EarthUnit, FireUnit, WaterUnit and AirUnit which we inherited from the Unit class. We implemented the common fields and functionalities in the Unit class and we customized the faction types where they differ in their own classes. For example, since all units heal and the healing logic doesn't differ among factions, we implemented the heal function in the base class. On the other hand, since the Air units are the only ones who can move, we implemented the calculating best possible move and sending that information in the AirUnit itself. A challenge we faced with this object oriented approach was, when we pass an object with mpi4py package, we actually pass a serialized copy of the object instead of a reference to that object. This made sense afterward since passing a reference actually means these workers can reach a shared memory but for inter-process communication, it is not the case. So, to overcome this duplication problem, we tried to be very careful while we are sending information about our objects (units in general) and not duplicating them without our control.

3.2 Regions

In the main body of our program, where we branch the execution into two based on the processor being the manager or worker, we create an instance of the Worker class we have implemented if the processor is not the manager. This worker class has the methods to simulate the changes and manage the units by communicating with the other workers and the manager. It has specialized methods to calculate the next state of its own grid and these methods return the changes which will happen in that phase in a structured way, similar to JSON which includes all the necessary information to understand the changes which will happen in that field. After calculating the changes which will happen for that phase in their own grid (Region2 + Region3), each worker shares the changes which their neighbors need to know with their adjacent workers. So, for example, if a worker controls a large grid, it doesn't need to inform its neighbors for a change which will happen around its center (Region3) as it won't affect their next calculations.

3.3 Message Passing

The main purpose of the project was utilizing message passing and we needed to be careful to not cause any deadlocks or overhead. While other strategies could work, we tried to come up with a logic which we think is reasonable and easy to implement. We first played with the MPI package to understand how the message passing methods (comm.recv() and comm.send()) works. We found out that when a processor calls the send method but not tries to receive anything, it can continue its execution without any problems even if nobody receives that message. On the other hand, when a processor tries to receive a message from another processor, it waits until it actually receives. After experimenting and understanding these properties, we decided to use this schema to implement inter-processor communications:

- When the manager and the workers need to communicate, the manager sends the messages to all workers by iterating over their ranks. And immediately after completing sending these messages, the manager starts to collect responses, iterating over the worker ranks once again. This way, when we ensure that each worker receives and sends a message to the manager in return, we don't cause any deadlocks and the processors continue with their own execution.

- Similarly, when the workers need to communicate among themselves, each processor first sends the messages to its neighbors and right after sending, each processor starts collecting the responses, from its neighbors once again. Once each neighbor for a processor complete the sending stage and starts receiving, we know that the message we want to get is already has been sent to us. So we don't wait for each other and the message we wait is guaranteed to be sent.

3.4 Simulating the Changes

As we have said earlier, each worker simulates and shares the changes with its neighbors, after which it receives the necessary information from the neighbors again. This way, after communicating with only the adjacent cells, each worker has the necessary information to conclude the moves. It starts applying the changes in its own field (Region1 + Region2 + Region3, Region1 being the neighbor cells). After making the changes and resolving the conflicts (like combining air units if they decided to move to the same

cell), each worker has the same and correct updated grid for their share, including the necessary neighboring worker cells. Note that workers don't calculate the actions which will be taken by the units which are located in their neighbor's grid, but they update their field according to the simulation results shared by their neighbors.

3.6 Communication Content

Lastly, let us talk about the "packets" shared between workers. The packets are the data structure we named to exchange information between workers and they carry the necessary changes' information to the neighboring workers like "from where to where by who" information while moving the air units or "from which unit to which unit with what attack power" etc. These packets are generated by each worker while calculating the actions in a phase and they are shared with the adjacent workers at the end of the phase. Each worker receives and resolves the actions in these packets to correctly update their regions with their own packet as well which they have also sent to the neighbors.

4 Partitioning Strategy

4.1 Strategy Used

We have used the Checkered Partitioning to achieve a more balanced workload. Although the communication overhead increases in checkered partitioning, we expect the checkered partitioning to outperform the striped partitioning on uneven but large input sizes.

The total number of communications for each processor is larger on average in the checkered partitioning as the boundaries between adjacent workers in total increase. However, since it outperforms the Striped Partitioning in uneven and large data sizes and provides better parallelism, we still prefer it.

4.2 Communication Between Processes

Since we keep the neighboring cells' territory synchronized, we pass the changes in our cells which affect the neighbors. We used "packets" to pass along the updates in a structured way instead of directly passing all the information. This way, we didn't have to exchange the unchanged cells' information. Each worker passes the updates on its outer cells to the adjacent workers and the adjacent neighbors update their regions accordingly. The processors in the middle of the board have 8 neighboring processors which are right, left, up, down and diagonals while the processors on the edges and corners have less. Each processor can reach to its neighbors' ranks by calculating them based on its own rank and each processor send the changes in its fields to the neighbors and each processor waits for its neighbors' updates.

4.3 Advantages and Disadvantages

Both methods have both advantages and disadvantages which make them suitable for various use cases. The Checkered Partitioning approach simply outperforms the Striped Partitioning approach when the data is unevenly distributed such that most of the units

are placed on the same row. Also, if the board size is (NxN) and processor number is also N^2, the Striped approach only enables the use of N processors since it assigns one processors to one row at maximum. However, in this case, Checkered Partitioning enables the use of all of N^2 processors, thus more parallelism. However, even though Checkered approach provides more parallelism and load balancing, it makes communication between neighboring processors harder, simply because the number of neighboring processors is much higher. Also, it is considerably harder to implement Checkered Partitioning, since the communication is more complex.

5 Test Results

Provide the results of your tests. Include examples of initial and final grid states and discuss performance metrics such as execution time and communication overhead.

5.1 Example Input and Output

Input:

```
8 2 2 4
Wave 1:
E: 0 0, 1 1
F: 2 2, 3 3
W: 4 4, 4 5
A: 6 6, 7 7
Wave 2:
E: 1 0, 2 1
F: 3 2, 4 3
W: 5 4, 6 5
A: 7 6, 0 7
```

Output:

```
Initial Board
---------------
E . . . . . . .
. E . . . . . .
. . F . . . . .
. . . F . . . .
. . . . W W . .
. . . . . . . .
. . . . . . A .
. . . . . . . A
---------------
```

```
Final Board
---------------
E . . . . . . .
E E . . . . . .
. E F A W . . .
. . F . W . . .
. . . F A W . .
. . . . W W A .
. . . . . W . .
. . . . . . A .
---------------
```

5.2 Performance Analysis

5.2.1 Execution Time

When we simulated the example input which is really small, we did not observe a significant difference in execution times. However, more worker counts seemed to give longer run times overall due to the communication overhead. Therefore, in order to see the benefits of parallelism, the program must be executed with larger input samples. Another important issue is that we couldn't use more than 4 more than 4 worker processors since our machines only had 8 processors but we expect the communication overhead to increase as the number of workers increases.

When we ran an artificial large input (1024 by 1024 board) which is not really homogeneous, we observed this output times between 1 worker and 4 workers.

```
mpiexec -n 2 main.py ./io/input1.txt ./io/output1.txt  233.33s user 3.77s
system 199% cpu ----> 1:58.86 total

mpiexec -n 5 main.py ./io/input1.txt ./io/output1.txt  187.47s user 3.92s
system 492% cpu ----> 38.876 total
```

As you can see, the multiple workers achieved a higher CPU utilization and the ratio between the run times are almost, as we expected, equal to the worker processor counts' ratio which is 1/4.

5.2.2 Communication Frequency

When we analyze how many times we make communications and among whom, we can say that we tried to keep the communication number minimal since it will be the limiting factor for the efficiency of the project. At the start of each wave, the total grid information is read and partitioned by the manager. After correctly formulating each worker's field, the manager sends the new fields to the workers. The manager, as we have said earlier, waits for the end of the wave after this step. The workers however, after receiving the wave information, start simulating the phases for each round. There are 4 different phases given in the project description which are movement, action, resolution and healing. Since we tried to keep the inter-process communication minimal, we combined the last 3 and we make 2 data exchanges between workers in each round, one for the movements which change the state of the grids, and the other for the attacks, healings,

removals of the dead units etc. Also, after simulating the wave and before ending the wave by returning to the manager, each worker again shares information for the last time in that wave in order to simulate the water units' special ability "flood".

For these variables, we can calculate the number of total inter-process communications.

n = number of processors ($a^2 + 1$), w = wave count, r = rounds each wave

- manager-worker communications: $2 \cdot w \cdot (n - 1)$
- worker-worker communications: $w \cdot 2r \cdot (8 \cdot (n-1) - 12\sqrt{n-1} + 4)$

As you can see, we can find an expression for the total number of communications based on the number of these 3 parameters and the complexity is proportional to each of them. The fastest execution times are expected to be found on relatively low number of processors but not a single processor since simulation which is ran by each processor separately starts to take significant time when the grid size gets really large.

6 Conclusion

This was the first project in which we used IPC and utilized parallelism. It enabled us to understand basic concepts of parallel programming and its strengths and challenges, such as communication overhead, and deadlocks. It was challenging and educative at the same time.

We used workers to simulate the "flood" ability of the waters at the end of wave. To achieve synchronization, we used message passing among the workers just like we did when we are simulating the rounds. So, one improvement to remove this last communication overhead, we could pass the water unit coordinates to the manager from each worker while we are ending the wave and let the manager decide the cells which will be flooded.

The point where we experienced the benefit of using MPI to interchange information was when we saw that we could actually pass objects and it automatically serializes and deserializes in the send and receive methods. This way, we were able to pass objects as a whole when we needed to.

References

- MPI Documentation: `https://mpi-forum.org`
- Python mpi4py Documentation: `https://mpi4py.readthedocs.io`