CMPE 300 - Analysis of Algorithms
Project 2 Report

Muhittin Köybaşı
2021400162

Osman Yusuf Tosun
2021400261

December 23, 2024

Contents

1 Introduction

The project simulates a battle among four elemental factions—Earth, Fire, Water, and Air—on an $N \times N$ grid-based battlefield using the Message Passing Interface (MPI). The simulation leverages parallel computing to model interactions between units, integrating various unit-specific attributes and abilities.

2 Design Decisions and Assumptions

Grid and Unit Structure

- The battlefield grid was implemented as a two-dimensional $N \times N$ matrix, where N is the size of the grid. Each cell in the grid can either be neutral (represented by a ".") or occupied by a unit belonging to one of the four factions: Earth (E), Fire (F), Water (W), or Air (A).
- Each unit was represented by a specialized class (e.g., `EarthUnit`, `FireUnit`), inheriting from a base `Unit` class. These classes encapsulate attributes such as health, attack power, and healing rate, along with methods for abilities specific to each faction.

Partitioning Strategy

- **Checkered Partitioning:** The grid was divided into square blocks, forming a checkerboard pattern. Each worker process was assigned one or more blocks.
- The number of workers was chosen such that N was divisible by the square root of the number of workers, ensuring equal-sized partitions.

Handling Unit Movements, Attacks, and Healing

- **Movement:**
  - Air units were allowed to move during the Movement Phase. Decisions were queued based on the number of attackable enemies in the vicinity after the move.
  - Conflicts during movement (e.g., two units attempting to move to the same cell) were resolved by uniting the units into a single stronger unit, as specified.
- **Attacks:**
  - Units attacked based on their faction-specific patterns. For instance, Earth units targeted adjacent cells (up, down, left, right), while Fire units could target all eight neighboring cells.
  - Damage was queued during the Action Phase and applied simultaneously during the Resolution Phase to ensure fairness.
  - Special abilities, such as Fire units' `Inferno` or Earth units' `Fortification`, were integrated into the attack logic.

- **Healing:**
  - Units that did not attack during a round healed based on their faction-specific healing rates.
  - Healing was implemented as an additive operation, capped at the unit's maximum health.

3 Implementation Details

The implementation of the project involved the following steps:

1. **Designing the Grid Structure and Partitioning:** The battlefield grid was implemented as an $N \times N$ matrix. A checkered partitioning strategy was chosen.
2. **Implementing the Manager Class:** The Manager was designed to:
   - Parse the input file to extract grid size, waves, and unit placements.
   - Divide the grid into partitions and distribute them to workers.
   - Coordinate simulation phases by sending commands (states) to workers.
   - Collect and aggregate results from all workers to generate the final output.
3. **Implementing the Worker Class:** Workers were responsible for simulating unit actions within their assigned grid partitions. They handled movement, attack, healing, and communication of boundary data with neighboring workers.
4. **State Descriptions:** The simulation involves several states that dictate the operations performed by workers. Each state is managed by the Manager and communicated to workers via MPI messages:
   (a) **State 1: Receive Grid Partition**
      - Workers receive their assigned grid partition from the Manager.
      - Each worker initializes its local grid and prepares for further instructions.

   (b) **State 20: Receive Units from Manager**
      - Workers receive unit data for their assigned grid partitions.
      - Units are placed in their respective positions within the grid.

   (c) **State 21: New wave**
      - Workers update the grid with new Water units generated during the Water flooding phase.
      - Clears the temporary set of newly created Water units.
      - Resets the inferno state for Fire units.

   (d) **State 2: Receive Boundary Data**
      - Workers receive boundary data from neighboring blocks.
      - This ensures that all units have accurate information about adjacent cells outside their partition.

   (e) **State 3: Send Boundary Data to Neighbors**

- Workers extract and send their boundary rows and columns to neighboring blocks.
- This data exchange ensures consistency for unit actions near partition edges.

(f) **State 4: Process Action Phase (Attacks)**

- Units decide whether to attack based on their health and available targets.
- Damage is calculated and queued for resolution in the next phase.

(g) **State 5: Apply Damage (Resolution Phase)**

- Queued damage is applied to all units simultaneously.
- Earth units reduce incoming damage by 50% using their Fortification ability.
- Units with health reduced to zero are removed from the grid.

(h) **State 6: Resolve Unit States**

- Applies accumulated damage to units and resolves their final health.
- Removes units from the grid if their health falls to zero.
- Earth units activate fortification if present.

(i) **State 7: Healing Phase**

- Units that did not attack in the previous phase heal based on their faction-specific rates.
- Healing is capped at the unit's maximum health.
- Resets the attack state for all units.

(j) **State 8: Water Unit Creation Phase**

- Water units convert adjacent neutral cells into new Water units.
- Expands the influence of the Water faction on the grid.

(k) **State 9: Water Unit Transfer Phase**

- Workers exchange information about newly created Water units with their neighbors.
- Ensures consistency of Water unit placement across boundaries.

(l) **State 10: Air Unit Movement Phase**

- Air units calculate their new positions based on attack opportunities and movement advantages.
- Units move within or across boundaries, as determined by the calculations.

(m) **State 11: Air Unit Transfer Phase**

- Workers handle the transfer of Air units that move across grid boundaries.
- Ensures that Air units maintain consistency in their new positions.

(n) **State 12: Air Unit Merge Phase**

- Resolves conflicts where multiple Air units attempt to occupy the same cell.
- Merges units into a single stronger unit, combining their attributes.

(o) **State 13: Send Grid State to Manager**

- Workers send their updated grid state to the Manager at the end of a wave.
- This data is used to aggregate the final state of the battlefield.

(p) **State -1: Termination**

- The Manager signals all workers to terminate execution.
- Workers perform any necessary cleanup and exit.

5. **Unit Classes and Abilities:** Custom classes for Earth, Fire, Water, and Air units were implemented with unique attributes and abilities. Special abilities such as Earth's fortification and Fire's inferno were integrated into the attack and resolution phases.

6. **Output Generation:** The final grid state was collected and written to an output file in the specified format after all waves were completed.

Challenges Faced and Solutions

1. **Synchronization and Deadlocks:** Managing communication between workers and the manager in a checkered partitioning strategy introduced complexities. Deadlocks were avoided by implementing an advanced even-odd communication scheme in both dimensions for a 2D grid *(see **4 Partitioning Strategy**)*.

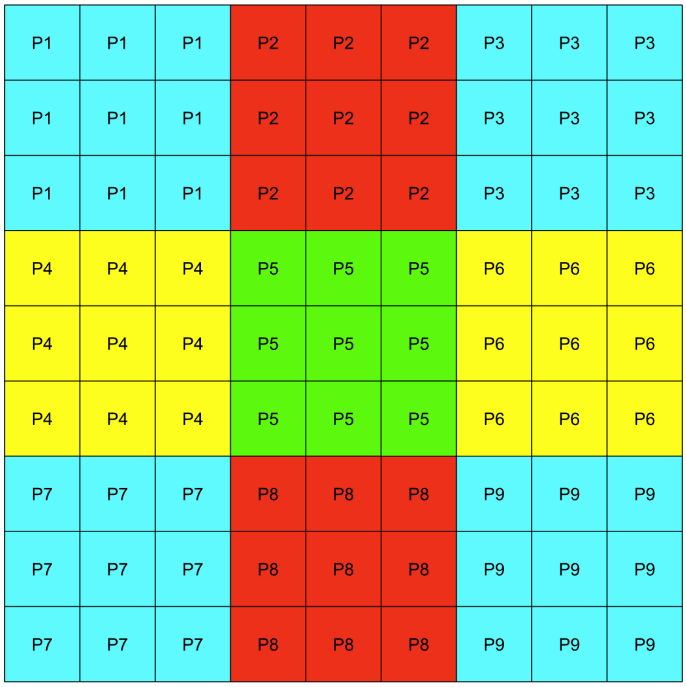
2. **Boundary Data Communication:** Exchanging boundary data between neighboring blocks required careful design to ensure correctness. A systematic approach was used to extract and update boundary cells efficiently. Each processor's neighbors and their relative positions are calculated by the manager before giving the blocks to each worker.

3. **Special Abilities Implementation:** Certain abilities, such as Air unit movement (Windrush) and Water unit flooding, were challenging to implement due to their dynamic nature. These were resolved by implementing queued decisions and applying changes simultaneously to avoid conflicts.

4. **Load Balancing:** Ensuring equal workload distribution among workers was critical for performance. Checkered partitioning was employed to balance workloads, and unit distribution was monitored to prevent imbalances.

5. **Decreasing Communication Overhead:** In the simplest implementation of checkered partitioning, neighboring processes exchange their entire boundary, which significantly increases communication overhead. To address this, during the implementation of the attack logic, a process only informs its neighbors about specific coordinates. While this approach reduces communication overhead substantially, it increases the total execution time since data exchange and attack calculations occur simultaneously.

4 Partitioning Strategy



4.1 Strategy Used

In this project, the checkered partitioning strategy was employed. This approach offers better theoretical complexity compared to sequential partitioning. When compared to striped partitioning, it provides a more balanced workload distribution among processors. However, it does not offer a significant advantage in terms of theoretical complexity, as the speedup is approximately proportional to the number of worker processors in both cases.

4.2 Communication Between Processes

In the checkered partitioning approach, each process requires boundary information from its neighboring processes. Specifically, a process needs data from up to 8 neighbors (up, down, left, right, and the four diagonals) before calculating the movement of air units. To achieve this, each process sends its boundary to its relevant neighbors which are precalculated by the manager. The size of the data being sent depends on the grid size and the total number of processes.

After calculating the units of flooded water, the updated unit data is exchanged between processes. The one-dimensional even-odd communication scheme is not suitable for checkered partitioning because a process typically communicates with 8 other processes unless it resides on the grid edge.

To address this, processes are divided into 4 main groups. In the above example, in a 9x9 grid divided among 9 processes, processes within the same group are represented by the same color. Each group exchanges boundary information with other groups. This communication strategy ensures there are no deadlocks, as a process either sends or receives data at a given time, but not both simultaneously.

4.3 Advantages and Disadvantages

In the checkered partitioning approach, more data is exchanged between processors, resulting in higher communication costs compared to striped partitioning. Additionally, the implementation is more complex, as calculating the boundaries to be sent and determining the appropriate neighboring processes to send the data to is more intricate. In striped partitioning, a process typically communicates with only two neighboring processes, whereas in checkered partitioning, a process may need to communicate with up to eight neighbors.

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

Example Inputs / *Example Outputs*

Example Input:

inputs > input.txt

```
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
```

Example Output:

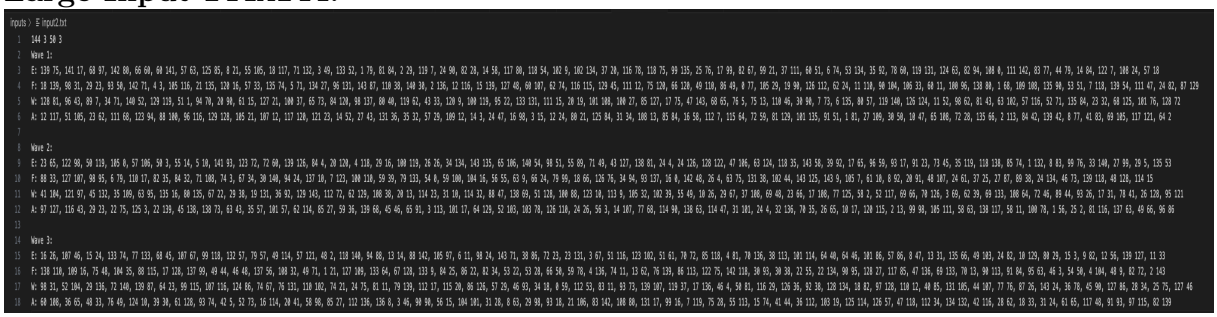
output.txt

```
E . . . . . . .
E E . . . . . .
. E F A W . . .
. . F . W . . .
. . . F A W . .
. . . . W W A .
. . . . . W . .
. . . . . . A .
```

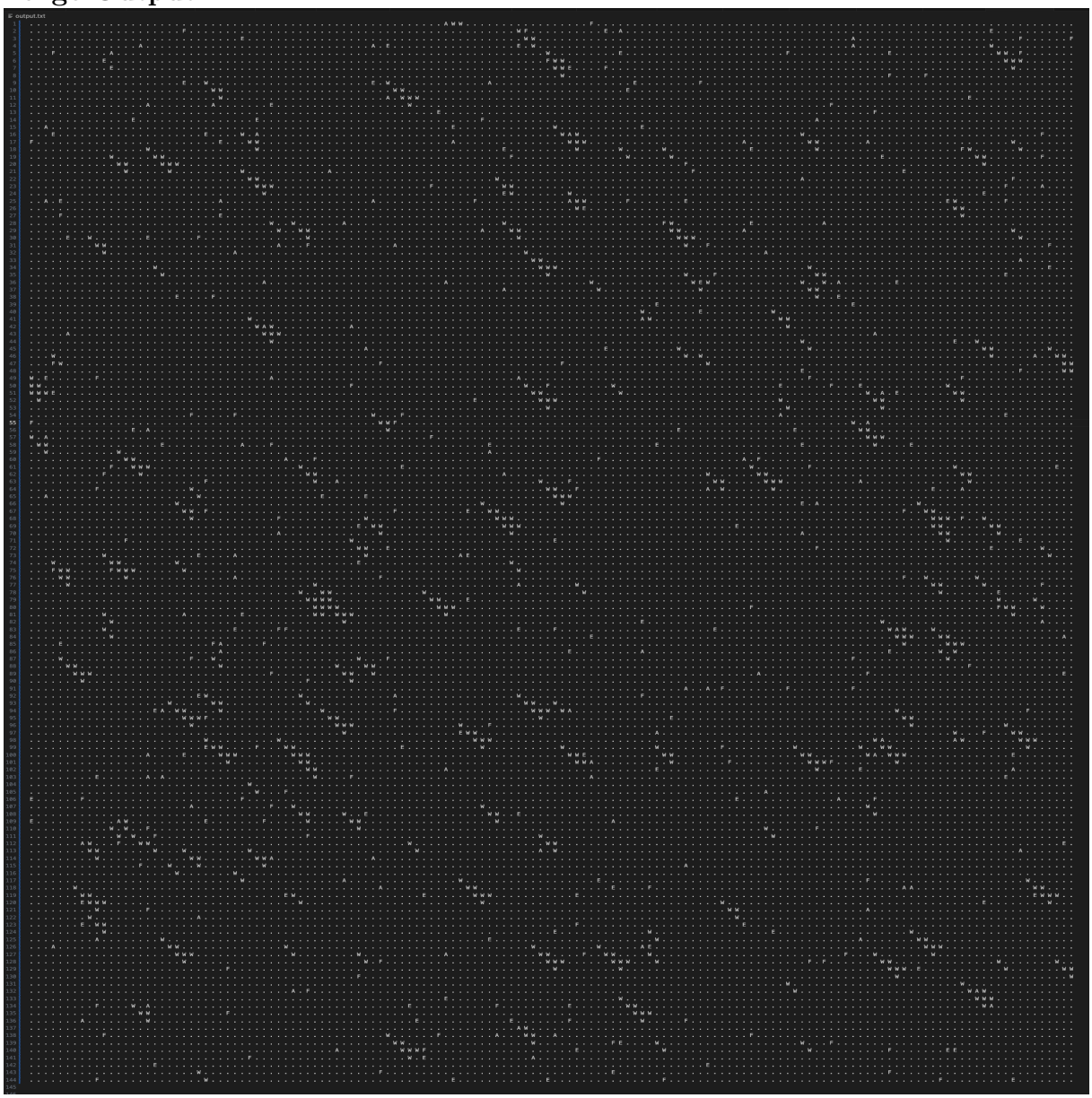
Execution Times:

```
(venv) → elements-parallel-chronicles git:(main) ✗ time mpiexec -n 2 python main.py inputs/input.txt output.txt
Rank 0 took 0.0025560855865478516 seconds to run.
Rank 1 took 0.0025610923767089844 seconds to run.
mpiexec -n 2 python main.py inputs/input.txt output.txt  0.13s user 0.06s system 89% cpu 0.207 total
(venv) → elements-parallel-chronicles git:(main) ✗ time mpiexec -n 5 python main.py inputs/input.txt output.txt
Rank 3 took 0.005199909210205078 seconds to run.
Rank 4 took 0.0051000118255615234 seconds to run.
Rank 1 took 0.005188941955566406 seconds to run.
Rank 0 took 0.005207061767578125 seconds to run.
Rank 2 took 0.005074262619018555 seconds to run.
mpiexec -n 5 python main.py inputs/input.txt output.txt  0.36s user 0.12s system 236% cpu 0.202 total
```

Large Input 144x144:



Large Output 144x144:



Execution Times 144x144:

```
(venv) ➜  elements-parallel-chronicles git:(main) ✗ time mpiexec -n 2 python main.py inputs/input2.txt output.txt
Rank 0 took 0.05758833885192871 seconds to run.
Rank 1 took 0.0575869083404541 seconds to run.
mpiexec -n 2 python main.py inputs/input2.txt output.txt  0.23s user 0.04s system 134% cpu 0.198 total
(venv) ➜  elements-parallel-chronicles git:(main) ✗ time mpiexec -n 5 python main.py inputs/input2.txt output.txt
Rank 0 took 0.07845115661621094 seconds to run.
Rank 1 took 0.07973408699035645 seconds to run.
Rank 3 took 0.07974076271057129 seconds to run.
Rank 4 took 0.07973599433898926 seconds to run.
Rank 2 took 0.07973289489746094 seconds to run.
mpiexec -n 5 python main.py inputs/input2.txt output.txt  0.79s user 0.10s system 299% cpu 0.298 total
(venv) ➜  elements-parallel-chronicles git:(main) ✗ time mpiexec -n 10 python main.py inputs/input2.txt output.txt
Rank 4 took 0.08237695693969727 seconds to run.
Rank 9 took 0.0829920768737793 seconds to run.
Rank 5 took 0.08177590370178223 seconds to run.
Rank 0 took 0.08313894271850586 seconds to run.
Rank 6 took 0.08276724815368652 seconds to run.
Rank 1 took 0.08192801475524902 seconds to run.
Rank 3 took 0.08195090293884277 seconds to run.
Rank 8 took 0.0829019546508789 seconds to run.
Rank 7 took 0.08303713798522949 seconds to run.
Rank 2 took 0.08180785179138184 seconds to run.
mpiexec -n 10 python main.py inputs/input2.txt output.txt  1.59s user 0.25s system 517% cpu 0.355 total
```

5.2 Performance Analysis

Theoretically, the program's speedup is proportional to the number of worker processes, as the workload is evenly distributed among them. However, the observed increase in execution time is primarily due to communication overhead. In a system specifically designed for parallel computing, the benefits of parallelism would outweigh the communication overhead. However, since this implementation was carried out on a system not optimized for parallel computing, the advantages of parallelism have not been significant.

6 Conclusion

Our checkered partitioning approach effectively distributed the battlefield workload among multiple workers, showcasing the potential for parallel speedup. However, communication overhead often overshadowed these gains, highlighting the need for optimized data exchange in highly parallel environments. Overall, the project underscored MPI's viability for grid-based simulations and its intricacies in balancing communication and computation.

References

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