CMPE 300 - Analysis of Algorithms
Project 2 Report
Rise of the Elements: The Parallel Chronicles

Name Surname 1 Student Number 1
Name Surname 2 Student Number 2

December 23, 2024

Contents

1 Introduction

In this project, we simulate a large-scale battle on an $N \times N$ grid among four factions—Earth, Fire, Water, and Air—each with its unique combat mechanics and special abilities. The simulation proceeds in waves, each wave comprising multiple rounds. Within every round, units may move (Air only), decide to attack or skip (thus potentially heal), and resolve all actions simultaneously. After all rounds are completed for a wave, certain factions (e.g., Water) have additional wave-level abilities (Flood) that can expand their influence on the battlefield.

Given the large size of the grid and the complexity of interactions, we employ the Message Passing Interface (MPI) to parallelize the computation. Each process (worker) handles a sub-portion of the grid, while the manager coordinates data distribution, result collection, and overall flow control.

In the following sections, we detail our design decisions, implementation, partitioning strategy, and test results.

2 Design Decisions and Assumptions

We made the following key design decisions and assumptions:

- **Grid Representation:**
  - We store the battlefield in a Python object we named Grid, which holds a 2D list of `Unit` objects (or a special `EMPTY_UNIT`), and also keeps track of information that will be frequently necessary (like the number of and the coordinates of different types of units). The manager process holds a NxN grid that contains all information of the battlefield, and each worker process holds a smaller nxn grid that only contains the information of their designated part of the battlefield.
  - Each cell can only contain one unit at a time, except Air units that can *merge* with other Air units upon moving to the same cell.
- **Unit Attributes and Mechanics:**
- Water's Flood ability looks for `EMPTY_UNIT` cells to convert to a water unit after each wave.
- Air units can move before attacking, possibly merging with another Air unit if they end up on the same cell.

Attack and Healing Logic:

- All damage is "queued" and resolved simultaneously; no single attack's result is visible before others are applied.
- Units that choose not to attack in a round will instead heal, but only if they survive.

MPI Manager-Worker Model:

- The manager (rank 0) reads the input file, broadcasts necessary data to the workers (rank 1 to $P-1$), and collects the results after each wave. After collecting the wave result, reads the next wave from the input file, incorporates them to the current state, and broadcasts the new state to the workers again. All communication needed during the wave is done among the workers without the intervention of the manager.
- Workers simulate the portion of the battlefield assigned to them and communicate with neighbors for boundary-related updates.
- At the end of all waves, the manager writes out the final battlefield state to the output file.

Assumptions for Simplicity:

- We assume the grid size N is divisible by $\sqrt{(P-1)}$ when performing checkered partitioning.
- We do not handle edge cases where the grid or P are not suitable for the chosen partitioning strategy (as per the project statement).
- We rely on the user to provide valid input (no out-of-bounds placements, no conflicting unit spawns). If conflicting unit spawns are given, for example, the spawns are ignored.

3 Implementation Details

Our code is primarily contained in the `main.py` file, using the `mpi4py` library to enable parallelism. Below is an outline of the main components:

1. **Parsing and Setup (Manager side):**
   - Read the parameters `N`, `W` (waves), `T` (units per faction), and `R` (rounds per wave).
   - Create a global `Grid` of size $N \times N$, and parse each wave's unit positions from the input.
   - Assign sub-grids to each worker according to our partitioning strategy (in this project, checkered partitioning).
   - Send these sub-grids and relevant parameters (`n` = size of sub-grid, number of rounds, etc.) to each worker.
2. **Simulation (Worker side):**
   - On receiving its assigned `Grid` portion, each worker processes `R` rounds:
     (a) **Movement Phase (Air Only):** Determine best positions for Air units, possibly merge if they move to the same cell.
     (b) **Action Phase:** Each unit checks if it should attack or heal. Attacks are "queued."

(c) **Resolution Phase:** Apply all damage simultaneously. Handle Earth's 50% damage reduction. Remove dead units; apply Fire's *Inferno* logic for kills.

(d) **Healing Phase:** Any unit that did not attack this round heals.

- After all rounds finish, **Water's Flood** expansions are performed: each Water unit attempts to spawn a new Water unit in an adjacent neutral cell.

3. **Data Exchange Between Neighbors:**

   - We implemented a function `communicate(...)` that handles sending and receiving data to and from the eight possible neighbor blocks in a checkered layout.
   - An "even-odd" or "A-B-C-D" labeling approach ensures no deadlocks occur; each sub-block has a send-first or receive-first logic, depending on its label.

4. **Final Merge (Manager side):**

   - After a wave completes, each worker sends its updated sub-grid back to the manager.
   - The manager updates the global `Grid`, proceeding to the next wave until all `W` waves are done.
   - At the end, the manager writes the final battlefield to `output.txt`.

4 Partitioning Strategy

4.1 Strategy Used: Checkered Partitioning

We chose **checkered partitioning** for this project. Specifically:

- We assume $(P-1)$ is a perfect square. We take $\sqrt{P-1} = k$ (integer) and thus divide the $N \times N$ grid into $k \times k$ blocks.
- Each sub-grid is of size $\frac{N}{k} \times \frac{N}{k}$, denoted in the code by $\texttt{n} = N/k$.
- Workers are assigned ranks from 1 to $P-1$; the top-left block corresponds to rank 1, the top-right block to rank k, and so forth, scanning in row-major order.

4.2 Communication Between Processes

In checkered partitioning, each worker can have up to 8 neighbors (top, top-right, right, bottom-right, bottom, bottom-left, left, top-left). We use these steps to avoid deadlocks:

1. We label each process as `A, B, C, D` based on its position in the $k \times k$ worker grid.
2. Depending on the label (derived from row-column parity), a process might send data first and then receive, or vice versa.
3. All boundary data (rows, columns, and corners) relevant to attacks or movements is exchanged via `MPI_Send` and `MPI_Recv`.

4.3 Advantages and Disadvantages

Advantages:

- Load tends to be more balanced across workers since each region is the same size.
- Particularly effective if units are spread out somewhat evenly across the grid.

Disadvantages:

- Communication is more complex since each block can have up to eight neighbors, increasing the overhead of boundary exchanges.
- Implementation complexity is higher compared to striped partitioning.

5 Test Results

5.1 Example Input and Output

Below is a simplified example showing how our program processes a small grid (`4x4`) with two waves and limited units. This is just a demonstration of the *format* rather than the full complexity of the project.

Sample Input (input.txt):

```
4 1 2 2
Wave 1:
E: 0 0, 1 1
F: 0 3, 1 2
W: 2 0, 3 1
A: 2 2, 3 3
```

Explanation:

- Grid size `N` = 4.
- `W` = 1 wave, each faction places `T` = 2 units, with `R` = 2 rounds per wave.
- Earth units at (0,0) and (1,1), Fire units at (0,3) and (1,2), etc.

Sample Output (output.txt) after all rounds and final expansions:

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

5.2 Performance Analysis

We tested our implementation on grids of size 16×16, 64×64, and 128×128 using 4, 9, and 16 workers respectively (plus 1 manager). Our primary observations:

- **Speedup:** For the 128×128 grid with 16 workers, we observed a noticeable reduction in total runtime compared to a single-process run.

- **Communication Overhead:** As expected with checkered partitioning, the overhead of communicating with up to 8 neighbors grows with P, but the improved load balance compensates for it in many scenarios.
- **Scalability:** The simulation scales better when each worker is assigned a sufficiently large sub-grid; extremely small sub-grids (too many workers for the problem size) increase the communication ratio.

6 Conclusion

This MPI-based simulation showcased the effectiveness of parallel computing for a grid-based battle scenario. By using a manager-worker model and carefully orchestrating simultaneous attacks and movements, we accurately represented the complex interactions of the four elemental factions.

- **Lessons Learned:** We gained hands-on experience managing boundary exchanges in distributed grids, especially in a scenario where each cell's update depends on multiple neighbors.
- **Potential Improvements:**
  - Streamline the communication protocol to reduce message size.
  - Implement advanced load-balancing for highly uneven distributions of units.
- **MPI Benefits:** Dividing the grid and distributing the workload simplified concurrency. MPI's flexibility allowed us to tune the partitioning scheme (striped or checkered) to balance communication overhead and load distribution.

Appendices

No additional appendices were required.

References

- MPI Documentation: `https://mpi-forum.org`
- Python `mpi4py` Documentation: `https://mpi4py.readthedocs.io`
- Project Description: Provided by course staff.