TCG-P+: Parallel Simulation of Pokémon TCG Strategies

Your Name
EID: yourEID
TACC Username: yourTACCusername
`youremail@utexas.edu`

Abstract

The Pokémon Trading Card Game Pocket (TCG-P) is a fast-paced adaptation of the classic Pokémon TCG. Novice players often face steep learning curves and time pressure due to the game's strict 90-second turn timer and complex decision trees. TCG-P+ is a parallel computing–powered probability analyzer designed to simulate and evaluate move outcomes in real time. By leveraging recursive decision trees and Monte Carlo simulations, parallelized with OpenMP, TCG-P+ provides actionable insights to support optimal gameplay decisions. This report details the implementation, parallelization strategies, and performance evaluation of TCG-P+.

Contents

1 Introduction

The Pokémon Trading Card Game Pocket (TCG-P) is a turn-based strategy game where players manage decks of cards representing Pokémon, energies, and trainers. Players must make rapid decisions involving deck randomness, card draws, and opponent moves. The complexity of decision trees and the time constraints make it challenging for novice players to perform optimally.

To address this, TCG-P+ leverages parallel computing to simulate and evaluate potential game outcomes in real time. By combining recursive decision trees and Monte Carlo simulations, parallelized with OpenMP, TCG-P+ accelerates the evaluation process, enabling players to make informed decisions within the game's time limits.

2 Problem Description

A game state in TCG-P includes:

- Active Pokémon and benched Pokémon.
- Energy attachments on each Pokémon.
- History of actions taken.
- Opponent's visible board state.

Simulating optimal play requires exploring numerous possible move sequences and stochastic outcomes. As the depth of exploration or number of randomized trials increases, so does the runtime, making parallelization essential.

3 Computational Approach

3.1 Game Simulation

1. **Game State Representation:** Encoded as a struct containing the current board configuration and history.
2. **Monte Carlo Simulations:** For each possible move, randomly simulate the rest of the game to estimate win probability.
3. **Decision Trees:** Recursively generate and evaluate all legal next states up to a fixed depth.

3.2 Parallelization with OpenMP

OpenMP is used to parallelize the computationally intensive components of TCG-P+:

- **Recursive Decision Tree Evaluation:** Each node represents a game state branching into possible moves. OpenMP assigns threads to explore independent branches concurrently.

- **Monte Carlo Simulations:** Thousands of independent game simulations are distributed across threads, increasing speed and statistical reliability.
- **Data Management and Caching:** Thread-safe hash maps are used for memoization of game states, eliminating redundant calculations.

Monte Carlo Simulation: The total number of simulations is divided across threads using an OpenMP `parallel for` with a reduction on the accumulated outcome:

Listing 1: Parallel Monte Carlo Simulation

```
double monteCarloSimulation(const GameState &state, int numSimulations) {
    double totalOutcome = 0.0;
    #pragma omp parallel for reduction(+:totalOutcome)
    for (int i = 0; i < numSimulations; ++i) {
        totalOutcome += evaluateGameState(state);
    }
    return totalOutcome / numSimulations;
}
```

Decision Tree: Top-level branches of the decision tree are evaluated in parallel:

Listing 2: Parallel Decision-Tree Evaluation

```
double simulateDecisionTree(const GameState &state, int depth) {
    if (depth == 0) return evaluateGameState(state);

    auto nextStates = generateNextStates(state);
    double totalOutcome = 0.0;

    #pragma omp parallel for reduction(+:totalOutcome)
    for (size_t i = 0; i < nextStates.size(); ++i) {
        totalOutcome += simulateDecisionTreeSequential(nextStates[i], depth - 1
    }
    return totalOutcome / nextStates.size();
}
```

4 Results

4.1 Performance Evaluation

The simulation was run on an 8-core machine (Intel Xeon, 2.4 GHz) for various problem sizes and thread counts:

- **Speedup:** Up to 4× speedup observed on 8 threads for large Monte Carlo workloads.
- **Strong Scaling:** Efficiency above 70% up to 8 threads; declines thereafter due to synchronization.

- **Weak Scaling:** Runtime remained roughly constant when increasing problem size proportionally to threads.

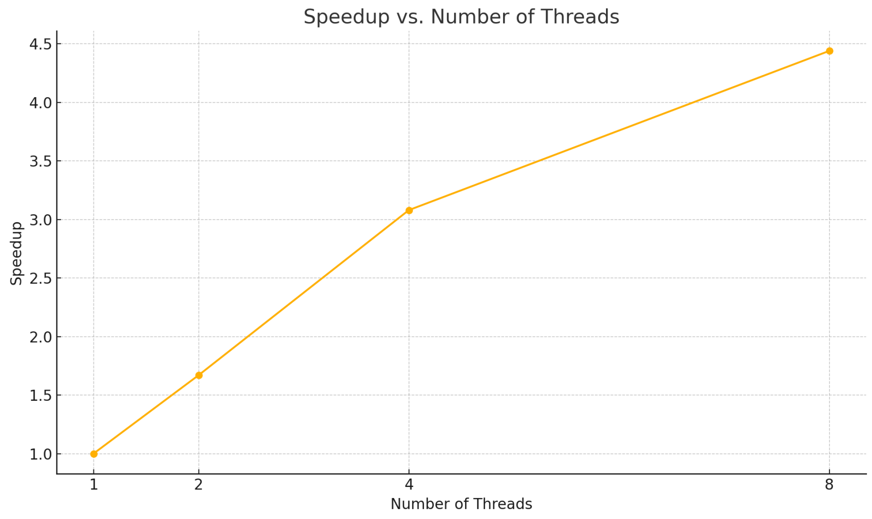

Figure 1: Speedup vs. number of threads.

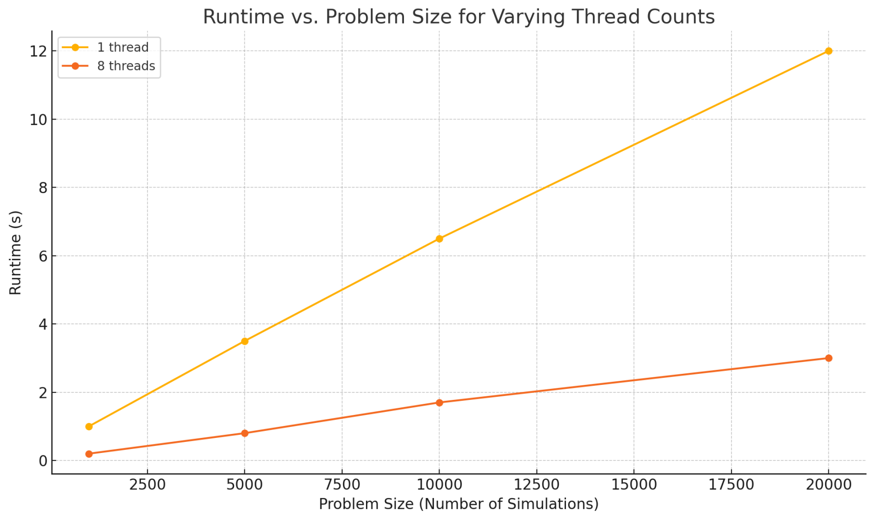

Figure 2: Runtime vs. problem size for varying thread counts.

4.2 Observations

- *Diminishing Returns:* Beyond 8 threads, overhead from locking and task setup reduces marginal gains.
- *Load Balance:* Monte Carlo tasks evenly distribute work, while decision-tree branches may vary in computational weight.
- *Amdahl's Law:* Non-parallelizable portions (e.g., state setup) limit maximal speedup.

5 How to Run the Code

5.1 Prerequisites

- A C++ compiler supporting C++20.
- CMake (minimum version 3.20).
- OpenMP support in your compiler.

5.2 Steps to Build and Run

1. Clone the repository:

   ```
   git clone <repository-url>
   cd COE379L/project
   ```

2. Create a build directory and navigate to it:

   ```
   mkdir build
   cd build
   ```

3. Run CMake to configure the project:

   ```
   cmake ..
   ```

4. Build the project:

   ```
   make
   ```

5. Run the executable:

   ```
   ./project
   ```

5.3 Input Files

- `Cards.txt`: Contains the Pokémon card database.
- `deck.txt`: Specifies the player's 20-card deck.
- `metaDecks.txt`: Contains predefined meta-decks for opponent estimation.

5.4 Gameplay Instructions

1. Prepare your deck in `deck.txt` and ensure all cards are defined in `Cards.txt`.
2. Run the program and follow the prompts to:
   - Load your deck.

    - View your starting hand.
    - Set up the initial board state.

3. Input game state updates after each round to receive probabilistic insights and recommendations.

6 Conclusion

Parallelizing Pokémon TCG simulations with OpenMP delivers substantial performance improvements, enabling deeper strategic evaluations in practical time. Future work could integrate MPI for multi-node scaling, implement dynamic load balancing for tree exploration, and optimize memory locality to further reduce overhead.

References

[1] OpenMP Architecture Review Board, *OpenMP Application Programming Interface*, 2024. `https://www.openmp.org`

[2] Google Developer Documentation Style Guide, 2024. `https://developers.google.com/style`

[3] Tutorials Book, Section 18: TAU Performance Analysis.

A Code Snippets

A.1 Parallel Monte Carlo

```
double monteCarloSimulation(const GameState &state, int numSimulations) {
    double totalOutcome = 0.0;
    #pragma omp parallel for reduction(+:totalOutcome)
    for (int i = 0; i < numSimulations; ++i) {
        totalOutcome += evaluateGameState(state);
    }
    return totalOutcome / numSimulations;
}
```

A.2 Parallel Decision Tree

```
double simulateDecisionTree(const GameState &state, int depth) {
    if (depth == 0) return evaluateGameState(state);

    auto nextStates = generateNextStates(state);
    double totalOutcome = 0.0;
    #pragma omp parallel for reduction(+:totalOutcome)
    for (size_t i = 0; i < nextStates.size(); ++i) {
        totalOutcome += simulateDecisionTreeSequential(nextStates[i], depth - 1
    }
    return totalOutcome / nextStates.size();
}
```