Parallel Programming Assignment 1

Name and Student ID: Kai-Wei Yen / b12902147

Implementation Details

The solver uses **A* search** algorithm. The implementation details are as follows.

1. Data Structure

- **map**: `std::bitset`
  The map is stored as multiple bitsets, where the number of bits corresponds to the map size. Each tile type has its own bitset, and a bit is set to 1 if the corresponding position contains that tile type.
- **position**: `int`
  The position is stored as a single integer which represents the index of the tile in the bitset (map). For example, the position `(r, c)` is stored as `r * ROWS + c`, where `ROWS` is the number of rows in the map.
- **state**: `struct State`
  - `player`: `int` — the position of the player.
  - `boxes`: `bitset` — the positions of all boxes.
  - Some other members for A* search and path reconstruction.

2. Heuristic Function

The heuristic function is defined as the sum of squared distances from each box to its nearest target.

$$h(state) = \sum_{b \in boxes} \min_{t \in targets} dist(b, t)^2$$

where $dist(b, t)$ is the distance (considering walls) between box b and target t.

3. Deadlock Detection

- **simple deadlock**: Precompute all the Simple deadlock and mark tiles that cannot be occupied by boxes as **fragile tiles**.
- **state deadlock**: During the search, if there exists a 2 by 2 square that contains only walls and boxes not on target, the state is considered a deadlock.

4. Solver (A* Search)

- A `std::priority_queue` is used to store the states, always expanding the one with the **lowest heuristic value**.
- A `std::unordered_set` keeps track of visited states to prevent revisiting.
- For each box, **check whether it can be pushed in each of the four directions**. If valid, generate a new state and insert it into the priority queue.
- A state is discarded if it represents a **deadlock** or has already been **visited**.
- When a state is found where all boxes are on target positions, the solution path is reconstructed and returned.

Optimizations

There are several optimizations implemented to improve the performance of the solver. The performance is represented by the output of `hw1-judge`.

1. BFS Solver to A* Search

The original solver uses BFS algorithm, which leads to exploring many unnecessary states. By switching to A* search (manhattan distances heuristic), the solver can prioritize states that are more promising, leading to faster solutions.
Performance Improvement: {19 17.53} → {20 16.55}

2. Combined States with Same Box Positions

In the original implementation, states with the same box positions but different player positions are treated as separate states. By combining these states, we can significantly reduce the number of states to explore.
Performance Improvement: {20 16.55} → {21 7.45}

3. Precompute Simple Deadlocks and More Powerful Heuristic

By precomputing simple deadlocks and using a more powerful heuristic (sum of squared distances), the solver can avoid exploring dead-end states and prioritize more promising states.
Performance Improvement: {21 7.45} → {24 50.73}

4. Memory Optimizations

Several memory optimizations have been implemented to reduce the memory footprint of the solver. These include:

- Using **state index** in the priority queue instead of the entire `struct State`.
- Using `std::bitset` for representing the map instead of a 2D vector.

Performance Improvement: {24 50.73} → {24 21.95}

Parallelization with OpenMP

There was an attempt to parallelize the **state expansion process** using **OpenMP**, however, it did not yield significant performance improvements.

OpenMP vs. `std::thread`

OpenMP was chosen instead of `std::thread` because it provides a **simpler and higher-level interface** for parallelizing loop-based workloads. Whereas, using `std::thread` can significantly **increase code complexity** despite its flexibility. In this project, OpenMP was a more practical choice for quickly experimenting with parallelization while maintaining **readability and maintainability**.

Parallelization Results

Due to the nature of the **greedy best-first search** algorithm and the overhead of managing shared data structures (such as the **priority queue** and **visited set**), the parallelization did not yield significant performance improvements. The **overhead of thread management and synchronization** outweighed the benefits of parallel state expansion.