HW1 Report

R13922213 梁正

1. Implementation Description:

This implementation solves a Sokoban puzzle using parallel breadth-first search (BFS) with OpenMP. The core algorithm employs level-synchronous parallelization where all states at each BFS level are processed in parallel.

Key components include:

- A Vertex structure representing game states with a flattened 2D map, player location, and dimensions
- Movement logic in try_push() that handles player movement and box pushing mechanics
- Comprehensive deadlock detection including corner deadlocks, wall deadlocks, freeze deadlocks, and pattern-based deadlocks
- Parallel BFS implementation using thread-local storage to minimize critical sections
- Custom hash function for vector<char> to efficiently store visited states
- Path reconstruction by maintaining parent pointers and move history

The parallelization strategy uses thread-local containers during state expansion, then merges results in a single critical section to reduce contention. Dynamic scheduling with chunk size 32 helps balance workload across threads.

2. Difficulties and Solutions:

The main challenges encountered were:

- Memory Management: Initially used malloc/free but switched to new/delete for proper C++ object construction/destruction. Implemented careful cleanup of all allocated Vertex objects to prevent memory leaks.
- Synchronization Overhead: Early versions had excessive critical sections causing contention. Solved by implementing thread-local storage patterns where each thread accumulates results locally before a single merge operation.
- Deadlock Detection Accuracy: Balancing pruning effectiveness with correctness was challenging. Conservative deadlock detection was implemented to avoid false positives that could eliminate valid solution paths. Multiple deadlock types were implemented but with careful validation.
- State Representation Efficiency: Originally used string conversion for hashing,

which was expensive. Implemented a custom hash function for vector<char> to improve performance while maintaining collision resistance.

- Load Balancing: Static scheduling caused uneven work distribution. Dynamic scheduling with appropriate chunk sizes provided better load balancing across threads as BFS levels can have varying computational complexity per state.

3. pthread vs OpenMP Comparison:

Strengths of pthread:

- Fine-grained control over thread creation, synchronization, and scheduling
- Explicit memory model and barrier synchronization capabilities
- Platform-independent thread management
- Better suited for complex producer-consumer patterns or irregular parallelism

Weaknesses of pthread:

- Higher programming complexity and verbose syntax
- Manual thread lifecycle management increases development time
- More prone to synchronization bugs like deadlocks and race conditions
- Requires explicit handling of load balancing and work distribution

Strengths of OpenMP:

- Simpler programming model with directive-based parallelization
- Automatic load balancing through various scheduling strategies
- Built-in reduction operations and private variable handling
- Easier to incrementally parallelize existing sequential code
- Excellent for loop-level parallelism and structured parallel patterns

Weaknesses of OpenMP:

- Less control over thread behavior and scheduling details
- Limited flexibility for complex synchronization patterns
- Fork-join model may not suit all parallel algorithms efficiently
- Debugging parallel code can be more challenging due to implicit thread management

For this Sokoban solver, OpenMP was the appropriate choice because the parallel BFS algorithm fits well with the fork-join model, and the built-in dynamic scheduling effectively handles load balancing across BFS levels. The thread-local storage pattern works naturally with OpenMP's thread identification mechanisms.