Implementation Overview

The solver is built on a bidirectional A* algorithm tailored for Sokoban. Two search processes are maintained in parallel: one expanding from the start state by pushing boxes, and the other from the goal state by pulling boxes. Each side uses a priority queue ordered by f = g + h, where g is the depth and h is a heuristic (e.g., Manhattan distances between boxes and targets). Expansion continues until the frontiers meet in the same normalized state, at which point a complete solution is reconstructed.

For parallelization, I implemented a batched multi-threaded expansion strategy. At each iteration, a batch of promising states is taken from the global queue and distributed among worker threads. Each thread processes its own subset, expanding successors and recording them in thread-local data structures (sub_queue, sub_visited). Once all threads complete, results are merged into the global pool.

To achieve this, I used pthread directly, rather than higher-level abstractions. This allows manual control over spawning and joining threads as well as explicit management of synchronization via mutexes. Although more verbose than frameworks like OpenMP, pthread gives fine-grained control over thread lifecycle and resource sharing, which was essential for handling the irregular workload of A* search.

Difficulties and Solutions

The main difficulty was synchronizing global structures (priority queues, visited maps, histories). If all threads wrote directly to shared data, lock contention would create significant overhead.

To address this, I applied a divide-and-conquer design:

- Each thread expands its own local pool of states independently.
- Results are stored in local visited maps and queues, avoiding global contention.
- At the end of each batch, threads synchronize, merging their local results back into the global structures.

This design minimized synchronization costs and kept parallel expansion efficient.

Strengths and Weaknesses of pthread and OpenMP

pthread

- Strengths: Offers low-level control over thread creation, destruction, and synchronization. Flexible enough to handle complex, irregular workloads like search trees.
- Weaknesses: Verbose, harder to debug, and requires careful manual management of locks and thread lifecycle.

OpenMP

- Strengths: Very convenient for parallelizing simple loops and data-parallel tasks with minimal code changes. Highly portable.
- Weaknesses: Less suitable for irregular control-flow tasks such as graph or A* search. Offers less flexibility for custom thread coordination.