8-Puzzle Solver Documentation

Angela Bojarski

Introduction

The 8-puzzle problem is a sliding puzzle consisting of a 3x3 grid with eight numbered tiles and one blank space. The goal is to move the tiles until they are in a specified goal configuration. This paper presents an analysis of a multithreaded solution to the 8-puzzle problem, mirroring the breadth-first search (BFS) idea and a hash table to track visited states.

Source Code Documentation

Overview

Source Files	Definition
8-puzzle-hash.c	Contains the main puzzle solver code
linked-list.c	Contains the declaration of the node data structure and corresponding operations

Data Structures and Operations

Shared_queue

Shared queue structure for managing nodes in the BFS algorithm. This queue is used by multiple thread to synchronize their work. It contains pointers to the head and tails of the linked list, mutex and condition variables for synchronization, count of nodes, and found state.

Hash_table

Hash table structure for storing visited nodes. This helps in checking if a state has already been visited to avoid redundant work. It contains an array of linked lists for handling hash collisions and a mutex for controlling access to the hash table.

Thread_arg_t

Structure for thread arguments passed to each thread in the search algorithm. It contains a unique thread identifier, pointer to shared queue structure, pointer to hash table of visited states, and the goal state of the puzzle.

Node

Node structure representing each state of the puzzle in the search tree. It contains the puzzle state, coordinates of the empty tile, node depth, pointer to next node in linked list, pointer to parent node in tree, array of pointers to child nodes, and number of children nodes.

Problem and Solution

Memory Leaks

Memory leaks occur when dynamically allocated memory is not properly deallocated, leading to unused memory that the program cannot reclaim. In the initial testing stages of the programs, it was observed from a valgrind test that significant amounts of memory were not being freed.

To address this issue, I ensured that all dynamically allocated memory is freed once it is no longer needed. Each node representing a puzzle state is created by a call to malloc and must be freed after it has been processed. Additionally, I implemented the free_tree function to deallocate the entire search tree once the solution is outputted and the free_table function to free all the linked lists in the hash table. Finally, I ensured that all my data structures were freed accordingly. This approach helps manage memory efficiently and prevents leaks.

Excessive Memory Usage

In the second phase of testing, I noticed that any puzzle depth greater than fifteen would result in the termination of the program. After some research I discovered that this was likely due to the improper addition of nodes to the shared queue, when the puzzle state had already been visited, resulting in redundant node creation.

To address this problem, I decided to implement a hash table for the thread to check if a puzzle state has already been visited and added to the queue. This ensures that a new node will only be allocated if it is unique. Implementing memory-efficient data structures and minimizing unnecessary memory allocations further helps in controlling excessive memory usage.

Eager Consumer

An eager consumer refers to a thread that waits for work to be available in the queue and processes it as soon as it arrives. In this scenario, if a thread is created and tries to process a node from an empty queue because another thread has not added any new nodes yet, it can lead to undefined behavior.

To manage the workload among threads, we use mutexes and condition variables to synchronize thread activities. Threads wait on the condition variable when the queue is empty and are signaled when new nodes are added to the queue. Proper use of mutexes and condition variables help maintain efficient and fair thread scheduling.

Long Critical Section

A long critical section occurs when a thread holds a mutex for an extended period of time, blocking other threads from accessing the shared resource. In the 8-puzzle solver, I noticed that the long critical section significantly degraded the performance by causing threads to spend excessive time waiting for the mutex, leading to reduced parallelism and increased contention for resources.

To mitigate the impact of long critical sections, I broke the critical sections up, minimizing the necessary operations performed when the mutex is held. I accomplished this through using a lock for the hash table and only grabbing it inside the is_visited and add_to_visted functions, and ensuring a thread only had the queue lock if they are actively adding or removing a node. By reducing the duration of mutex locks, I allowed other threads to access shared resources more frequently, improving overall performance.

Test Case

To test the accuracy of the code, the output was compared to that of a correct program using BFS in Python.

The initial state puzzle provided was:

```
//Inital puzzle state
int start[N][N] = {
    {8, 7, 6},
    {5, 4, 3},
    {2, 1, 0}
};
```

The final state puzzle provided was:

```
//Goal puzzle state
int goal[N][N] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 0}
};
```

The correct solution to this problem was 30 steps.

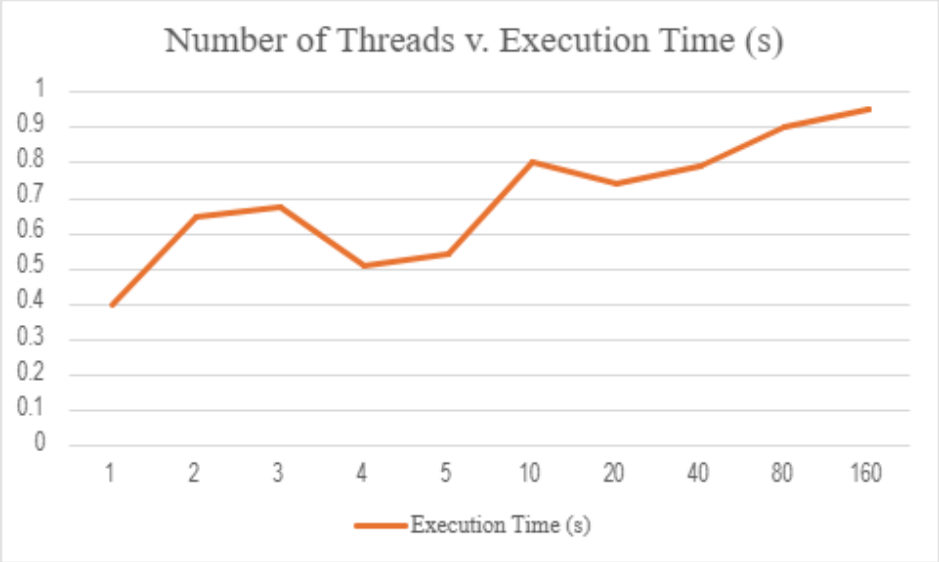
Performance Comparison

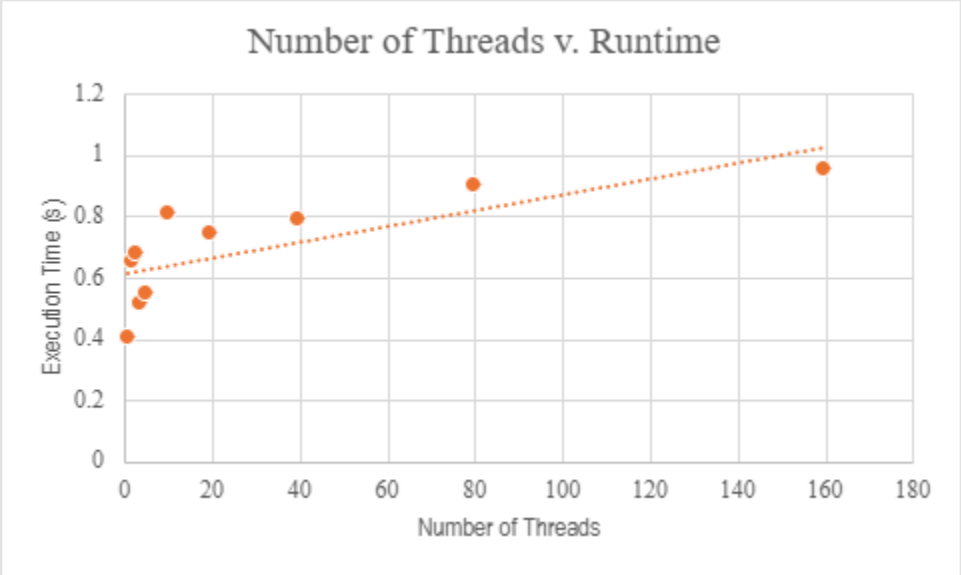
The performance of the multithreaded 8-puzzle solver was recorded with a varying number of threads. The results are summarized in the table below:

Number of Threads	Execution Time (s)
1	0.400
2	0.647
3	0.674
4	0.512
5	0.542
10	0.803

20	0.740
40	0.798
80	0.901
160	0.950

The best performance was in a single thread, with an execution time of 0.4 seconds. Beyond that, the execution time increased with the number of threads, indicating diminishing returns on parallelism.





Limitations

Correctness of BFS Execution

The current implementation does not use condition variables to ensure that the children of a node are added to the queue in the precise order of their generations. This can lead to scenarios where nodes are processed out of the expected BFS order. Without guaranteeing the correct order of node processing, the algorithm might deviate from the pure BFS behaviors.

Overhead of Thread Management and Synchronization

The solution's performance is also limited by the overhead of thread management and synchronization. As the number of threads increases, the contention for shared resources also increases, leading to performance bottlenecks, diminishing the benefits of parallelism. The time spent waiting for mutexes as more threads are added ends up increasing the running time of the program.

Future Work

Ensuring BFS Order

Implement condition variables to enforce the correct order of node processing, maintaining the integrity of the BFS algorithm.

Conclusion

While the multithreaded 8-puzzle solver performs faster than its Python counterpart, we did not observe any speed-ups in the execution time as the number of threads increased. Even with the mentioned problem and solutions, we actually observed an increase in the execution time as the number of threads increased. This is likely due to the cost of the overhead in managing the threads and shared resources, like the mutex blockings.