15-Puzzle Algorithm Parallel Report

Juliette Hainline (juliettehainline) and Forrest Sill (forrestsill)

December 8, 2016

1 Introduction

We implemented the A* algorithm to solve the 15 puzzle. In this report, we will outline our methodology, data structures, and go into detail about our algorithm for the serial implementation. We will then go into detail about how we parallelized this algorithm for the second half of the project.

2 Methodology for Serial Implementation

Data Structures

1. A **priority queue** that contains boards that have not yet been tested.
2. A **vector** that contains all boards that have already been tested.

Algorithm

1. First, the inputted board is loaded into the priority queue as the first element.
2. All boards that could possibly result from a move on the original board (either 2, 3, or 4 boards, depending on whether the free space is on an edge, a corner, or neither) are loaded into the priority queue.
3. The original board is popped from the priority queue added to the vector of already-tested boards.
4. This process repeats with the board at the top of the priority queue (that is, the board that has the lowest priority score, which means it is estimated to be the closest to the desired board configuration.)
5. This continues until the board at the top of the queue matches the desired board configuration.

Priority Score

The priority score of a given board is a measure of how close the given board is to the desired configuration. It is calculated by adding the board's manhattan score with its hamming score. Its **manhattan** score is calculated by summing the distance from each block to its correct location using only horizontal and vertical moves. For example, a block that is in a location diagonal from its correct location would have a manhattan score of two. A board's **hamming** score is simply the number of blocks that are in the wrong location. For example, if the board is correct except for two blocks that are flipped, its hamming score would be two. Through experimentation we were able to find some minor improvements to the classic A* algorithm for our parallel implementation.

3 Methodology for Parallel Implementation

To implement our algorithm in parallel, we used the pthreads API, a C++ library. We wanted to create 4 threads as specified in the writeup. Therefore, we took the first 4 boards that result from four of the first possible moves. We implemented the algorithm as outlined above on each of these four possibilities with different threads. To ensure that threads would not modify the three shared data structures (closed, won, and winningThread) simultaneously, we used mutexes.

4 Comparison of Serial and Parallel Implementations

When we ran both our serial implementation as well as our parallel implementation using 4 cores on linux.cs.uchicago.edu, we saw an average increase in solve speed of 56.1% for the puzzle in input.txt.

15 Puzzle Serial vs. Parallel Implementation on input.txt			
Run Number	Serial Speed (seconds)	Parallel Speed (seconds)	Percent Increase
1	0.568	0.239	57.9%
2	0.534	0.228	57.3%
3	0.563	0.283	49.7%
4	0.531	0.222	58.2%
5	0.532	0.221	58.4%
6	0.532	0.223	58.0%
7	0.531	0.222	58.2%
8	0.532	0.234	56.0%
9	0.564	0.273	51.6%
10	0.531	0.232	56.3%
Average	**0.542**	**0.238**	**56.1**%

Why Parallel Speedup Is Not 4x (Amdahl's law)

If our serial implementation of the A* search algorithm was perfectly parallelizable between the four cores, we would have expected to see a 75% improvement in speed. We didn't, and we attribute that to two points:

1. **A* search is not perfectly parallelizable.** We rely on a vector that contains all already-examined game boards. This vector can only be modified by one thread at a time, therefore other threads stop execution while waiting for the mutex. Overcoming this hurdle would be a major speed improvement, as this algorithm accesses this data structure *very* often. A solution would be to implement a data structure that can be modified by multiple threads at once. This should be possible, because the only modification that is done is adding elements to the vector, not overwriting or changing the order of already-added elements.

2. **Our implementation does not split the work perfectly evenly between the 4 cores.** Each of the four cores receives one of the first 4 boards, but not all boards can result in the same number of future boards. For example, two boards (that each go to one core) may be very close to each other and would therefore have many overlapping moves. These two threads would finish faster than a thread that started with a board that was farther away (because already-examined boards are not examined twice.) We did not add a load-redistribution system, but that would be a good way to mitigate this deficiency. For example, as soon as one thread finishes its execution (assuming, of course, that a solution has not yet been found), it would take one half of the remaining boards from a still-executing thread.

Amdahl's law tells us that the higher the parallelizable proportion of a program, the better able more processors will be able to improve execution time. Making improvements to both of the above points could significantly increase the proportion of our program that is parallelizable and, therefore, execution time.