Parallel Computing Term Project - Optimal Rubik’s Cube Solver

Venkata Sai Ruthvick Kommineni

**I. Introduction**

For the project, I parallelized an Optimal Rubik’s Cube Solver. The serial algorithm that i started with was originally created in 1997 by Michael Reid, a professor at the University of Central Florida. I had originally planned to implement my own serial algorithm, but quickly realized that the complexity and time requirement was outside the scope of this project. I had considerable difficulty finding suitably simple, open source, C code for the project, but we eventually found an updated 2004 version of Dr. Reid’s algorithm.

This program gives the “optimal” solution to a scrambled Rubik’s cube, meaning that it gives the solutions to the cube that involves the least amount of moves possible. Upon startup, the program initializes a pattern database, referred to as a distance table in the code utilized.

When run on the computers of the time the code was originally written, this initialization period took around 11 minutes. However, when i ran the program on the ECS cluster, it took about 13 seconds due to dramatic increases in processor speeds over the last two decades. Once the pattern database is initialized, the user can input a scrambled cube for the program to solve. In this implementation, a solved cube is represented as a sequence of “cubies,” which are the small cubes that make up the Rubik’s cube. There are 26 cubies, with nine on each of the three layers minus the middle layer, which does not have a center cubie. The solved cube is represented as follows: UF UR UB UL DF DR DB DL FR FL BR BL UFR URB UBL ULF DRF DFL DLB DBR.

The letters in each cubie represent the faces that it is on - U is Up, D is Down, F is Front, B is Back, R is Right, L is Left. Edge cubies are represented by the two faces they span, while corner cubies are represented by the three faces they span. Center cubies are not included because they are only on one face, so it would be redundant to include them in this notation. A scrambled cube is given by enumerating each cubie on the scrambled cube that corresponds to each cubie in the solved representation. An example of a scrambled cube is: UF UR RD RB LU LF DB DL FR UB DF BL UFR RFD RDB RBU LFU LUB DLB LDF.

To find the solution, the program performs a specialized depth-first called iterative-deepening-A\* or IDA\*, which “looks for increasingly longer solutions in a series of iterations” (Korf). Using a depth-first algorithm allows for finding the shortest possible sequence of moves to solve the cube. The median number of moves for optimal solutions of scrambled cubes is 18, with a proposed theoretical limit of 26 moves. I found the majority of solutions to be between 16 and 19 moves. The number of nodes created increases sharply after 16 moves, which results in a large discrepancy between the time to find a 16-move solution and a 19-move solution. Sixteen-move solutions averaged about 3 seconds, while 19-move solutions averaged over 40 seconds. I focused on two particular solutions to individual scrambled cubes, one which was 16 moves and the other which was 19 moves.

For the 16-move solution, a total of 1348 tests were performed, with 1702038 nodes created. For the 19-move solution, at a depth of 18 we had 54350 tests and a total of 272192594 nodes created. I was not able to obtain information from depth 19 because of the structure of the program, but we were able to see the dramatic increase in computations even at a depth of 18 compared to 16.

The program outputs the solution as a sequence of moves by giving the face that needs to be rotated. The solution to the scrambled cubie given above is: F R' B U L' B L F L F' L' B' R' F' R B' R F'

The “prime” marks (‘) indicate a counterclockwise move; otherwise a clockwise move is assumed. The user can continue to input cubes for solving, using the same pattern database created at the start of the program, until they choose to exit.

My research has shown that prior parallel implementations of Rubik’s Cube solvers are limited, especially optimal Rubik’s Cube solvers. The main source of our information, a 2008 paper on parallelization of a Rubik’s Cube solver, discusses the parallel implementation of a divide-and-conquer technique that does not involve the use of pattern databases. I found through my research and eventually through experience that the use of pattern databases does very well for efficient serial algorithms, but does not transfer very cleanly to parallelization. Therefore, I was not able to look at prior parallel implementations of our project, and therefore used what we have learned in the course so far to figure out which parts of the program are parallelizable and how to implement parallelization.

**II. Design**

I used POSIX Threads in C to parallelize this program. I used a set number of threads (four) because of the complex structure of the serial program and the limitations of parallelizable sections. I called my Pthreads function threadWork, and I passed in the arguments by using a struct. I anticipated having to pass multiple arguments to the function, but ended up only passing in the rank of the thread.

I began our parallelization by focusing on the creation of the pattern tables. This process was encapsulated in one function, init\_globals, that I placed in the Pthread function, threadWork, which I called upon creation of the threads. This process turned out to be relatively straightforward, but I was initially stuck on this section because of what appeared to be platform-specific issues. I was developing using Windows, and when I attempted to put the init\_globals function call into the Pthreads function, I received a Segmentation Fault. I was not able to figure out the cause of this issue, and i eventually tried running it on Linux (virtual machine, Ubuntu 16.04 LTS), which worked without any errors. From there, I began by splitting up the function calls for the initialization between the four threads, which involved setting the characteristics and bounds of the tree. i then inserted a barrier, using two semaphores (count\_sem and barrier\_sem), as provided in an example earlier this semester. This ensured that all threads waited to move on to inserting values into the tree until every thread was finished setting the characteristics of the tree.

After the barrier, I began initializing the distance table, which was relatively easy to split among threads. The init\_distance\_table function involves forward and backward searching through the data structure that adds nodes until it is complete. This searching is performed using two for loops, one for forward searching and one for backward searching. I divided the for loops among threads equally by dividing the loop size by the number of threads and assigning each thread an equal amount of work. On occasion, the last thread would end up with slightly more work if the number of iterations was not divisible by the thread count.

I was limited by how much I could parallelize this process. The algorithm involved updating and checking the neighbors of each node, so if i split the process among too many threads, it would result in reading data that had not yet been created or initialized. As a result, I found it best to manually control the number of threads I used, rather than allow that to be changed by the user (e.g. at the command line).

After finishing the parallelization of the pattern database creation, I moved on to the parallelization of solving the cube. I was originally under the impression that the pattern database creation took much longer than solving the cube. The manual included with Dr. Reid’s serial algorithm discussed the long startup time given by this process, and I had found that this process took longer than solving the cubes that I had tried entering. However, i found out that, as mentioned in the problem description above, the time for solving different cubes is extremely variable. When I found this out, I was able to see that while many cubes are solvable in a much shorter time that the pattern initialization takes, there are also many that take much longer than the pattern initialization. Upon this discovery, I decided to attempt to parallelize this section of the code as well in an attempt to make the times more consistent and especially reduce the times of the longer solutions.

By the time I started attempting to parallelize the solving mechanism, the project time window was coming to a close. Due to the early issues with the init\_globals function on Windows, and our misconception about the efficiency of the solve\_cube function, I did not allocate enough time for parallelization of this section of the code. i was able to get it running through a call in the threadWork function, but was limited to assigning all of the tasks to one thread. The structure and function-call hierarchy of this method was especially complex, and it is likely not possible to split among threads as it is. I would have likely had to completely restructure the algorithm that searches through the pattern databases. In order to parallelize a depth-first search, i would need to create subtrees for each thread; however, the IDA\*. The algorithm used by this program makes this very difficult, so we would probably have to change the searching algorithm used. However, this would also result in a loss of serial optimization, which could slow down the process so much that we would not obtain any speedup; additionally, some of the properties of the IDA\* are essential to finding the optimal solution, so i would need to ensure that i do not lose those properties in restructuring. It would be interesting to compare the performance of this optimized searching method to a more simple method in parallel.

**III. Implementation**

I have included both the serial and parallel versions of the program for testing.  The program must be built and run on a Linux machine.

**Serial**

To build the serial program from COMP137\_PROJECT directory (command line):

1. $ cd serial
   1. Enters the serial folder
2. $ gcc -Wall -O2 optimal.c
   1. Builds the project
3. $ ./a.out
   1. Runs the program
   2. When prompted to enter a cube, here are some options:
      1. UF UR RD RB LU LF DB DL FR UB DF BL UFR RFD RDB RBU LFU LUB DLB LDF
      2. UF UR RB RD LU LF DB DL FR UB DF BL RFD UFR RDB RBU LFU LUB DLB LDF
      3. UR UF RB RD LU LF DB DL FR UB DF BL RFD UFR RDB RBU LFU LUB LDF DLB
   3. Program will output how long it took to solve the cube
4. $ cat timing.txt
   1. Displays timing data

**Parallel**

To build the parallel program from COMP137\_PROJECT directory (command line):

1. $ cd parallel
   1. Enters the parallel folder
2. $ gcc -pthread -Wall -O2 optimal.c
   1. Builds the project
3. $ ./a.out
   1. Runs the program
   2. When prompted to enter a cube, here are some options:
      1. UF UR RD RB LU LF DB DL FR UB DF BL UFR RFD RDB RBU LFU LUB DLB LDF
      2. UF UR RB RD LU LF DB DL FR UB DF BL RFD UFR RDB RBU LFU LUB DLB LDF
      3. UR UF RB RD LU LF DB DL FR UB DF BL RFD UFR RDB RBU LFU LUB LDF DLB
4. $ cat timing.txt
   1. Displays timing data

**IV. Experiments**

I ran the program on ECS cluster for timing data and used the timer.h file that was provided earlier in the semester. i timed the initialization of the pattern database by calling getWorldTime directly before and after the call to init\_globals, which holds the hierarchy of function calls to create the entire table. i assigned the values of these two times to double variables, then printed out the difference between these two times to a text file for data accumulation. i performed the same operation with the serial and parallel algorithm, and in the parallel algorithm i had only thread 0 print out to the file so that i did not get duplicated data.

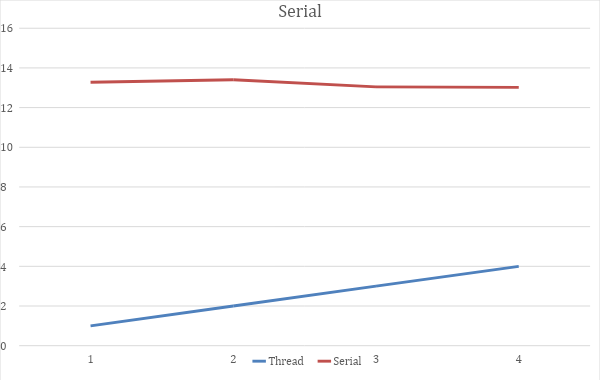
i also obtained timing data for cube solving, but because I was not able to parallelize this part, it does not lend any insight into the speedup, efficiency, or scalability of the problem. However, it does give us an idea of the significance of finding the shortest path, as well as the dramatic increase in computation time as solutions reach more than 16 moves.

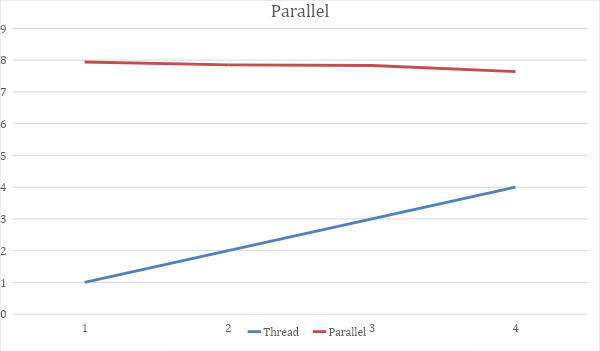
In my timing data, I included the startup time for Pthreads (i.e., pthread\_create), but this was somewhat insignificant considering the small amount of cores and the relatively large timing values that we were working with.

In order to replicate the experiments shown below, all one must do is just build and run the program (instructions for building and running included in Section III ). The program will output the timing data to a file called timing.txt, and the results can be seen there.

**Serial vs. Parallel: Pattern Table Initialization/Start Up Time (4 Threads)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Execution Time (s) | | | | |  |  |
| Threads | 1 | 2 | 3 | 4 | Average | Speedup | Efficiency |
| Serial | 13.277700 | 13.405296 | 13.042536 | 13.016887 | 13.185605 | 1.68692 | .421730 |
| Parallel | 7.942263 | 7.852035 | 7.831448 | 7.639751 | 7.816374 |





**V.Analysis**

From my experiment we can see that we were able to obtain a reasonable speedup of about 1.69 with our parallel implementation, but with four threads we have a low efficiency of .42. Such a low efficiency with very so little cores indicates that the program is not scalable, but it is difficult to make a definitive statement on the scalability of a program just given this information. However, Iam confident that the program is neither strongly nor weakly scalable because of the limited opportunities that the serial implementation offers for parallelization. A redesign of the serial algorithm is necessary to make this program scalable. In this current implementation, nodes within the pattern tree use information from their neighbors, which makes the typical implementation of splitting the tree into subtrees impossible. The cube-solving portion of the code is similar in that it prevents splitting the tree up. Therefore, the first step in future improvement is to find a new serial algorithm, extensively adjust the one I have, or create a new one. Now that we have a better understanding of how these types of programs work, we could likely create our own “parallel-friendly” program that allows for more straightforward implementation of parallelization.

**VI. Sources**

Korf, R., Finding optimal solutions to Rubik’s cube using pattern databases, aaai.org

Kunkle, D., Cooperman, G., Harnessing parallel disks to solve Rubik’s cube. Journal of Symbolic Computation (2008), doi:10.1016/j.jsc.2008.04.013