Investigating a Multithreaded Solution To a Rubik's Cube Solver

By

Ben Dickinson

exploring and comparing different techniques for solving a Rubik's cube challenge. This report will discuss the discrepancies of Multi and sequential threading in solving a complex Rubik's cube problem.

Demo video: https://youtu.be/NM-2cmlqTa4
Github link: https://github.com/benD02/SIT315---Final-Project.git

September 2023

SIT315 Deakin University

Abstract

The Rubik's Cube, an iconic and challenging puzzle, has captured the fascination of puzzle enthusiasts and computer scientists alike. In this report, we explore the task of solving a Rubik's Cube using two distinct approaches: multi + distributed threading and sequential threading. The objective is to evaluate the efficiency of these two methods and determine which is better suited for the complex task of solving the Rubik's Cube.

The Rubik's Cube presents an intriguing problem due to its vast number of possible configurations. Its solution requires the execution of a series of complex algorithms, and the exploration of various search spaces. We aim to address the question of whether parallelism, specifically through multi + distributed threading, can significantly accelerate the solving process compared to the traditional sequential approach.

Investigation and experimentation

To study the differences between these possible solutions, we will implement a few distinct solving algorithms. The first utilizes a sequential threading technique, allowing for a single flow of processes which occur one after the other. The second, a multi-threading technique, enabling concurrent processing of cube states on a single machine. And the third employs distributed threading, allowing multiple machines to collaborate on the solving process through a networked environment. In this experiment, we aim to measure and compare the performance of these approaches in terms of solving time and scalability.

- *hypothesis*

  In our study, we postulate that the sequential implementation will exhibit exponential growth in runtime as the problem size scales up. Conversely, for the parallel implementation, we anticipate a more linear increase in runtime as the problem size grows. Furthermore, in the case of the MPI + parallel implementation, we expect to observe a more pronounced increase in runtime, surpassing that of the purely parallel approach, due to the additional overhead introduced by distributed computing.

Methods and Resources:

Programming Environment:

The Rubik's Cube solving project was developed using the Microsoft Visual Studio development environment, utilizing the C++ programming language. Visual Studio provides a comprehensive integrated development environment (IDE) with features that streamline code development as well as efficient debugging applications.

Parallelization Framework:

The project will utilize the Microsoft Message Passing Interface (MPI) library to implement both multi-threading and distributed computing approach. MPI is a widely used library for developing parallel and distributed applications. It facilitates communication and coordination among multiple processes, making it well-suited for tasks that can benefit from parallelism across multiple computing resources.

Multi-Threading Approach:

For the multi-threading (Parallel) approach, the project employed C++'s native multi-threading capabilities, making use of the <thread> library. Threads were created to concurrently explore different branches of the Rubik's Cube solution space, thereby taking advantage of the multiple cores available on the CPU.

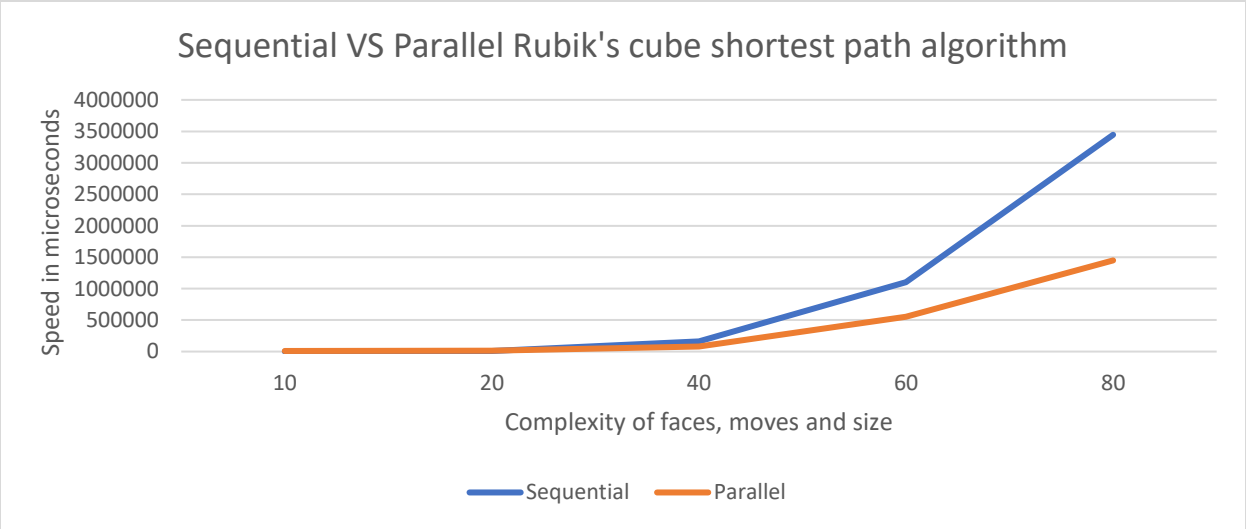
Performance Evaluation:

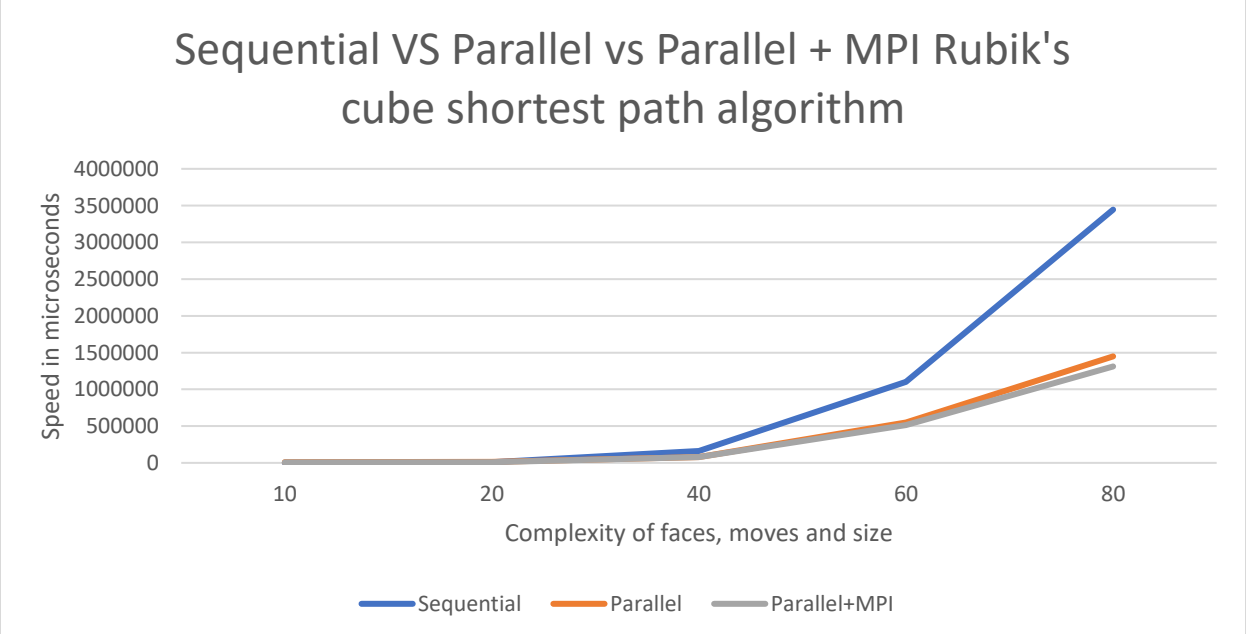
To assess the efficiency of the implemented methods, the project will conduct a systematic performance evaluation. A runtime metric will be declared to assess the efficiency of the implementation, the speedup, and scalability. These evaluations will provide insights into the advantages and limitations of each approach, helping to draw conclusions regarding their effectiveness in creating the best Rubik's Cube puzzle solver.

Results

In order to determine the optimal solution for solving Rubik's Cube, I devised three distinct mock-up programs: a sequential program, a parallel program, and a parallel + MPI program. These programs were designed to address various complexities of the Rubik's Cube problem, featuring different combinations of faces, moves, and sizes.

Utilizing a fundamental matrix system, where numerical values ranging from 1 to 5 corresponded to distinct cube colors, each program systematically tackled the Rubik's Cube challenge. These simulations served as a critical foundation for our comparative analysis, enabling us to gain valuable insights into the efficiency and performance of each solution approach under varying levels of complexity.





Discussion of results

The presented results provide a comprehensive overview of the runtime performance for the three different algorithms – Sequential, Parallel, and Parallel + MPI – as they are applied to Rubik's Cube solving tasks of increasing complexity. The graph displays the execution times in microseconds (μs) for each algorithm on the y axis, and representing the Rubik's Cube data complexities ranging from 10 to 80 units on the x axis. Example, a complexity of 40 equals a 40 faced cube, 40 x 40 cube size and 40 max moves. (just for testing purposes, this is not accurate to a real-world problem, just a metric used to test the cube's efficiency)

Sequential Solution:

The Sequential algorithm demonstrates an exponential increase in runtime as the complexity of the Rubik's Cube problems grows. As indicated by the data, solving a Cube with a complexity of 80 requires significantly more time (approximately 3,445,109 μs) compared to simpler configurations. This aligns with the initial hypothesis that the Sequential approach would exhibit exponential runtime growth.

Parallel Solution:

In contrast, the Parallel algorithm showcases a more linear increase in runtime as the problem complexity escalates. This suggests that parallelization effectively distributes the workload across multiple processing units, resulting in a more predictable and scalable performance. Solving a Cube with a complexity of 80 still requires more time than simpler problems but does not exhibit the same steep increase observed in the Sequential approach.

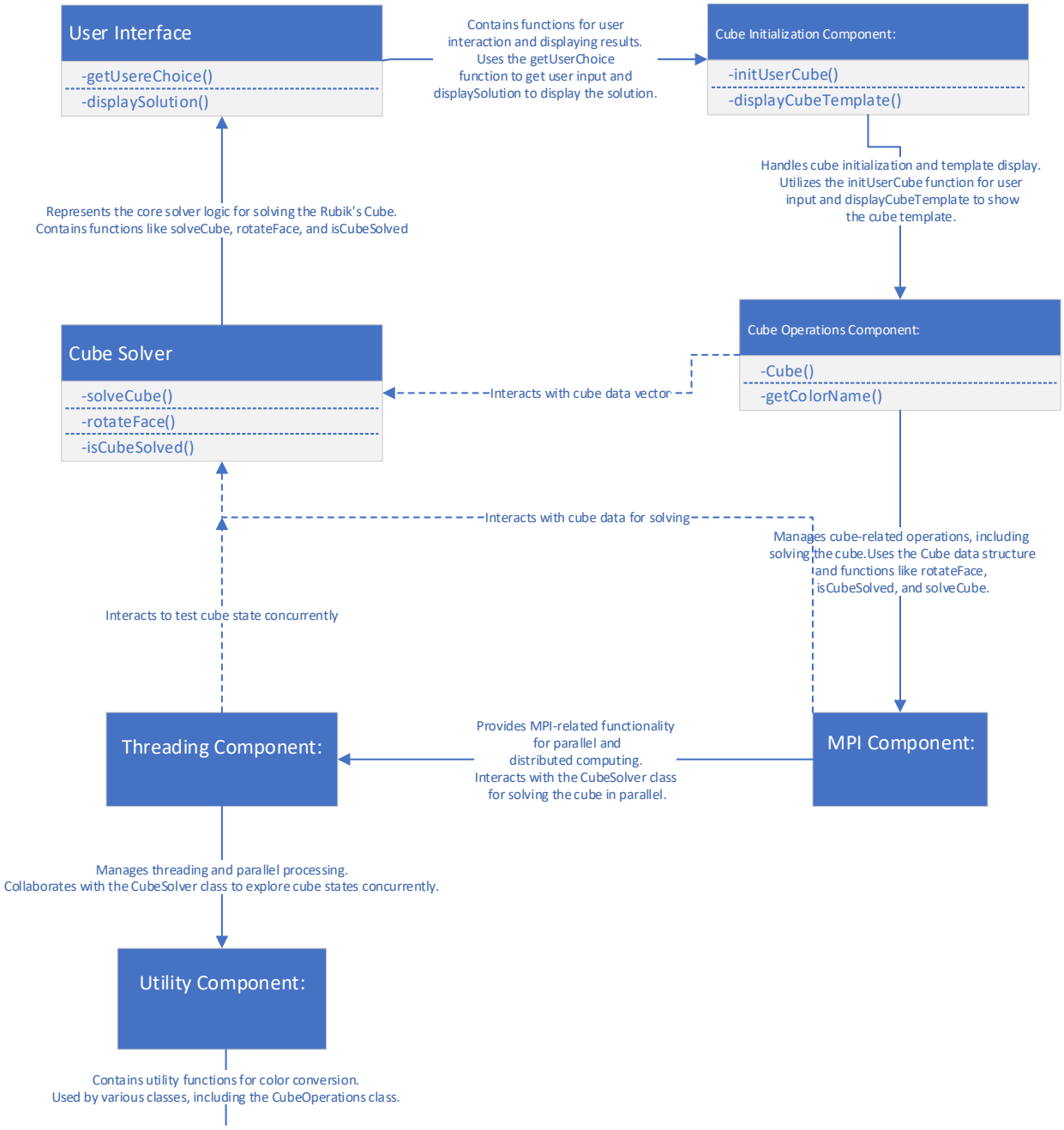
Parallel + MPI Solution:

The Parallel + MPI algorithm introduces distributed computing capabilities using MPI (Message Passing Interface). While it offers parallelism benefits, the inclusion of distributed computing introduces additional communication overhead, causing a slight increase in runtime compared to the pure Parallel approach. Solving the most complex Rubik's Cube problem in this configuration (complexity 80) requires approximately 1,311,727 μs, signifying a substantial performance penalty compared to the purely parallel solution.

In summary, these results underline the advantages of parallelization in managing the increasing computational load as problem complexity rises. The Sequential algorithm's

exponential runtime growth becomes a significant bottleneck for complex tasks. However, the introduction of distributed computing via MPI, while extending the problem-solving capabilities, does impose additional overhead. The choice of the most efficient algorithm depends on the specific problem complexity and available computational resources, emphasizing the trade-offs between sequential, parallel, and parallel + MPI approaches in Rubik's Cube solving. For this project I am going for the Parallel + MPI Solution as it is slightly quicker with the more complex solutions.

high-level code diagram



Documentation of final design

1. **Cube Colors**: The code begins by defining constants for different cube colors: WHITE, YELLOW, RED, ORANGE, GREEN, and BLUE. These constants are used to represent colors on the Rubik's Cube.
2. **getColorName Function**: This function takes an integer representing a color and returns a string representing the color's name. It's used to convert color integers into human-readable names.

```cpp
// Define cube colors
const int WHITE = 0;
const int YELLOW = 1;
const int RED = 2;
const int ORANGE = 3;
const int GREEN = 4;
const int BLUE = 5;

std::string getColorName(int color) {
    switch (color) {
    case WHITE: return "WHITE";
    case YELLOW: return "YELLOW";
    case RED: return "RED";
    case ORANGE: return "ORANGE";
    case GREEN: return "GREEN";
    case BLUE: return "BLUE";
    default: return "UNKNOWN";
    }
}
```

3. **Cube Struct**: This **struct** defines the structure of a Rubik's Cube. It uses a 3D vector to represent the cube's faces, allowing for variable dimensions based on the cube's size.
4. **initUserCube Function**: This function initializes a cube with user-defined colors. It takes the cube's size as input and prompts the user to input colors for each face of the cube.

```cpp
// Define a cube
struct Cube {
    std::vector<std::vector<std::vector<int>>> faces;  // Use a 3D vector for variable dimensions
};

// Function to initialize a cube with user-defined colors
Cube initUserCube(int size) {
    Cube cube;
    cube.faces.resize(6, std::vector<std::vector<int>>(size, std::vector<int>(size)));

    for (int face = 0; face < 6; ++face) {
        std::cout << "Enter colors for Face " << face << ":\n";
        std::cout << "Key: (0 - WHITE, 1 - YELLOW, 2 - RED, 3 - ORANGE, 4 - GREEN, 5 - BLUE)\n";

        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                int color;
                std::cout << "Enter color for row " << i << ", column " << j << ": ";
                std::cin >> color;

                if (color < 0 || color > 5) {
                    std::cout << "Invalid color entered. Please enter a number between 0 and 5.\n";
                    --j;  // Decrement j to re-enter the color for this cell.
                }
                else {
                    cube.faces[face][i][j] = color;
                }
            }
        }
    }
    return cube;
}
```

5. **rotateFace Function**: This function performs a clockwise rotation of a cube face. It takes the cube, the face to rotate, and the size of the cube as input. The function returns a pair containing the face number and the rotation direction ("LEFT" or "RIGHT").
6. **displayCubeTemplate Function**: This function displays the Rubik's Cube template, showing the colors on each face of the cube.
7. **isCubeSolved Function**: This function checks if a given cube is solved. It verifies that all cells on each face of the cube have the same color.

```cpp
// Function to perform a clockwise rotation of a cube face
std::pair<int, std::string> rotateFace(Cube& cube, int face, int size) {
    std::vector<std::vector<int>> temp = cube.faces[face];
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            cube.faces[face][i][j] = temp[size - j - 1][i];
        }
    }
    std::string rotationDirection = (size % 2 == 0) ? "RIGHT" : "LEFT";
    return std::make_pair(face, rotationDirection);
}

// Function to display the Rubik's Cube template
void displayCubeTemplate(const Cube& cube) {
    int size = cube.faces[0].size();

    // Display the cube template
    for (int face = 0; face < 6; ++face) {
        std::cout << "Face " << face << ":\n";
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                std::cout << getColorName(cube.faces[face][i][j]) << " ";
            }
            std::cout << "\n";
        }
        std::cout << "\n";
    }
}
```

```cpp
bool isCubeSolved(const Cube& cube) {
    int size = cube.faces[0].size();
    int numFaces = cube.faces.size();

    // Check each face to see if all cells have the same color
    for (int face = 0; face < numFaces; ++face) {
        int firstColor = cube.faces[face][0][0];
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                if (cube.faces[face][i][j] != firstColor) {
                    return false; // Cube is not solved
                }
            }
        }
    }

    return true; // Cube is solved
}
```

8. **solveCube Function**: This is the main cube-solving function. It takes the initial cube state and the number of threads as input. The function uses multi-threading to explore possible cube states and solve the cube. It also uses MPI for parallel processing.
    - It initializes MPI and obtains the process rank.
    - It divides the cube's search space among MPI processes based on their ranks.
    - It creates a queue (**q**) for breadth-first search (BFS) to explore cube states, along with queues (**stepQueue** and **rotationQueue**) to store the steps and rotations at each state.
    - It creates a thread pool with the specified number of threads.
    - Each thread explores cube states from the queue, checking if the cube is solved. If solved, it stores the solution and notifies other threads.
    - After all threads finish, the main process gathers solutions from other processes and determines the best solution based on specified criteria. It then broadcasts the best solution to all processes.

9. **getUserChoice Function**: This function displays a menu for the user to choose whether to solve the cube or exit. It returns the user's choice.
10. **displaySolution Function**: This function displays the solution steps, including the face number and rotation direction, in a human-readable format.

```
int getUserChoice() {
    int choice;
    std::cout << "Rubik's Cube Solver Menu:\n";
    std::cout << "1. Solve the cube\n";
    std::cout << "2. Exit\n";
    std::cout << "Enter your choice (1/2): ";
    std::cin >> choice;
    return choice;
}

// Function to display the solution steps, including the direction of rotation
void displaySolution(const std::vector<std::pair<int, std::string>>& solution) {
    std::cout << "Solution Steps:\n";
    for (int i = 0; i < solution.size(); ++i) {
        int faceNumber = solution[i].first;
        std::string rotationDirection = solution[i].second;
        std::cout << "Step " << i + 1 << ": Rotate face " << faceNumber << " " << rotationDirection << "\n";
    }
}
```

11. **main Function**: The main program starts by initializing MPI and obtaining the process rank. It repeatedly displays a menu for the user to choose an action:
    - If the user chooses to solve the cube, it prompts for the cube's size and initializes the cube with user-defined colors.
    - It calls the **solveCube** function to solve the cube using multi-threading and MPI.
    - If the cube is solved, it displays the solution steps. If not, it indicates that the cube cannot be solved.
    - The program continues until the user chooses to exit.
12. The program finalizes MPI before exiting.

This code provides a framework for solving a Rubik's Cube using parallelism and user-defined colors. The cube-solving logic can be further customized based on specific algorithms and criteria for solving the cube.

The process of the code:

1. The program starts by asking the user to choose an option. The user selects option 1 to solve the cube.
2. The user is prompted to enter the size of the Rubik's Cube, and in this example, a 2x2x2 cube is chosen.
3. The program then prompts the user to enter colors for each face of the cube using the key provided.
4. After entering the colors, the program displays the initial state of the cube.
5. The cube is solved, and the program displays the solution steps, indicating which face to rotate and in which direction.

Conclusion

Our investigation has demonstrated multiple ways to solve a Rubik's Cube using different threading techniques. The choice between sequential, parallel, or parallel + MPI approaches hinges on the specific problem's complexity and the computational resources at hand. It's essential to strike a balance between efficient parallelism and the potential overhead of distributed computing when tackling this captivating puzzle.

As we conclude this exploration, we emphasize the overall nature of computational problem-solving. The Rubik's Cube program served as a challenging yet interesting problem for testing and comparing various threading methodologies, and it gave me much insight into the exploration of efficient and scalable solutions in parallel computing.

References

[1] Rakshith. MG, "7 Rubik's Cube Algorithms to Solve Common Tricky Situations," Hobby Lark, 1 June 2023. [Online]. Available: https://hobbylark.com/puzzles/Rubik-Cube-Algorithms. [Accessed September 2023].

[2] Modus Beke, "Rubik's Cube Solver," rubiks-cube-solver, [Online]. Available: https://rubiks-cube-solver.com/. [Accessed September 2023].

[3] Chengzhi Huang. Tiane Zhu, "Rubik's ParaCube: a Collection of Parallel," Github pages , 2020, December.